Fluxion: A Differentiable DSP Runtime from Training GPU to Real-Time Edge

Matteo Spanio
Centro di Sonologia Computazionale (CSC)
Dept. of Information Engineering, University of Padova
Padova, Italy
spanio@dei.unipd.it

Antonio Rodà
Centro di Sonologia Computazionale (CSC)
Dept. of Information Engineering, University of Padova
Padova, Italy
roda@dei.unipd.it

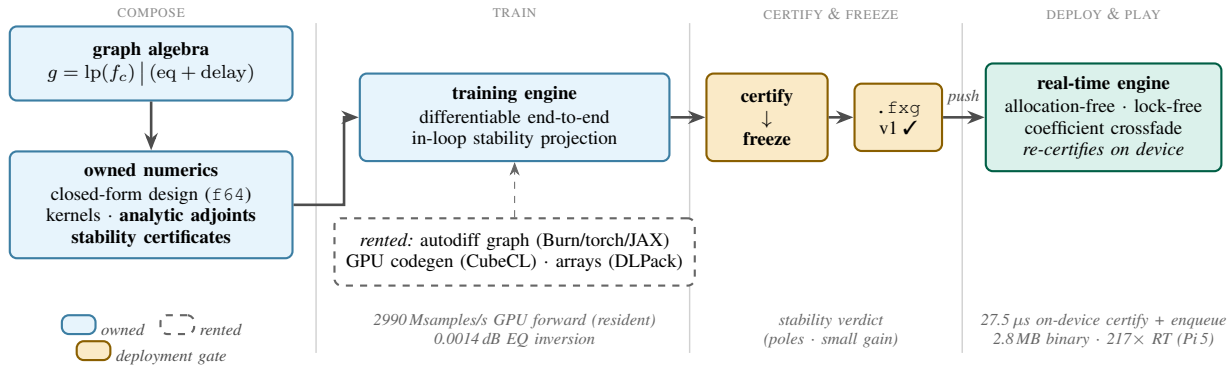

Fig. 1. Architecture overview. A graph algebra over Fluxion's own numerical code is lowered to a training engine and a real-time engine; the autodiff graph, GPU code generation, and array interchange come from existing libraries. A trained graph is checked for stability and frozen into a versioned artifact, which a Raspberry Pi-class device checks again before crossfading it in.

***Abstract*—Native real-time audio engines meet audio deadlines but are hard to connect to machine-learning workflows, while tensor frameworks integrate with machine learning but process audio as offline batches without timing guarantees; runtime layers built on them inherit the host's scheduling and deployment footprint. This paper presents Fluxion, a Rust audio DSP runtime for cascaded second-order sections, FIR filters, and delay-line effects. Fluxion implements the parts that are specific to DSP: closed-form filter design, forward kernels, analytic adjoints for the recursion, and stability checks. It relies on existing libraries for the autodiff graph, GPU code generation (tested here on CUDA), and array interchange. One graph algebra is lowered to a differentiable training engine and to an allocation-free real-time engine; between them, a trained graph is checked for stability and frozen into a versioned artifact that an edge device checks again before playback. We compare Fluxion with six framework-hosted and native baselines under one float32 protocol. Among the systems that compute the exact recursion, Fluxion has the highest GPU throughput when data stays on the device and the highest throughput on most CPU workloads, while its CPU training step is slower than those of torchaudio and GRAFX. Its test suite checks the audio thread for allocations with an instrumented allocator, and we report measured latency tails. Fluxion is open source at https://github.com/matteospanio/fluxion.**

***Index Terms*—audio DSP, DDSP, GPU computing, runtime systems, real-time audio, edge computing, Internet of Sounds**

I. Introduction

Native real-time engines such as Faust [1], SuperCollider [2], and JUCE [3] run audio with low latency, but they are hard to connect to the machine-learning tools now common in audio research. Tensor frameworks such as torchaudio [4], [5], nnAudio [6], and GRAFX [7] work well with machine learning, but they treat audio as offline batches of tensors, with no streaming state and no latency bound. Trained tensor graphs can be deployed on edge devices through TFLite or ONNX Runtime, but a sample-recurrent filter reaches those runtimes only by unrolling.

TorchFX [8], our earlier system, and differentiable-DSP libraries such as FLAMO [9] and ADAC [10] build a runtime layer on top of a tensor framework. TorchFX, for example, records composition as a deferred graph, fuses second-order-section (SOS) cascades into single dispatches, and reuses the host's autograd. These layers remain limited by their host: the interpreter schedules the real-time path, so the latency tail can be measured but not bounded, and the Python and PyTorch stack weighs gigabytes, which makes edge deployment impractical. The Internet of Sounds [11] makes this gap concrete: its Sound Things are networked embedded audio devices, and a filter trained on a GPU must run on them within audio deadlines and a small footprint.

We argue for a different division of work. A DSP runtime should *own* the parts that are specific to DSP: closed-form coefficient design, forward kernels, analytic adjoints for recursive filters, and the stability checks that decide whether a trained filter can be deployed. It should *rent* what general-purpose ML infrastructure already provides: the autodiff graph (Burn [12], PyTorch, or JAX), GPU code generation (CubeCL [13]), and zero-copy array interchange (DLPack [14]). Keeping the adjoints in the runtime lets several autodiff frameworks use them, and relying on existing libraries for the rest keeps the runtime small.

We present Fluxion, a Rust audio DSP runtime built this way (Fig. 1). A functional graph algebra (series `|`, parallel `+`, feedback `~`, and named nodes) is lowered to two engines: a training engine that composes analytic vector-Jacobian products (VJPs), and a lock-free real-time engine that does not allocate on the audio thread. Feedback is not yet lowered to either engine, so our results cover cascades, FIRs, and delay-line effects. Filters are designed in double precision; the single-precision coefficients that run are checked for stability and frozen into a versioned artifact, which the real-time engine plays.

Contributions.

1) A runtime architecture in which one graph IR is lowered to a training engine and an allocation-free real-time engine, with an explicit freeze step between them (Sec. III).
2) Hand-derived adjoints for the filter recursion, registered as custom operators in existing autodiff frameworks and extended to *design parameters* (cutoff, gain, Q), with a stability projection that keeps raw-coefficient training inside the stable region (Sec. III-B, Sec. IV-B).
3) A deployment pipeline that designs filters in `f64`, checks the stability of the frozen `f32` coefficients (pole radius per section, small gain for feedback loops), and stores them in a versioned artifact that the real-time engine loads without a design step (Sec. III-C).
4) A lock-free real-time engine with coefficient crossfades, whose tested audio-thread paths make no allocations under a counting global allocator, with a control test that confirms the counter works (Sec. III-D, Sec. IV-C).
5) A float32 comparison with exact-recurrence and frequency-sampling baselines on throughput, training cost, latency, and footprint, and an industrial case study (Sec. IV-A to IV-C).

II. Related Work

Tensor-framework audio. torchaudio [4], [5] provides exact time-domain recursion over batched tensors but no fused-cascade primitive, which Sec. IV-A measures. nnAudio [6] provides trainable GPU spectral front-ends without recursive filtering, so we do not use it as a comparator. GRAFX [7] composes differentiable audio graphs on PyTorch with an exact and an approximate filtering backend, and we measure both. Engel et al. [15] established the idea of learning through signal-processing structure (DDSP). All of these systems inherit their host's execution model.

Differentiable DSP. IIR filters can be made differentiable in two main ways. One is to build modules inside a host framework and use its autograd, as in FLAMO's frequency-sampled modules [9] and ADAC's FLAMO-to-FAUST compilation pipeline [10]. The other is to derive the adjoint of the recursion analytically and register it as a custom operator. This has been done for linear dynamical operators [16], all-pole filters [17], and direct-form filters [18], and torchaudio's `lfilter` and TorchFX's opt-in `torchfx.ddsp` module [8] also ship analytic backward passes. Fluxion takes the second approach and keeps the adjoints in the same runtime that checks and deploys the trained filter. Tab. III compares the cost of both approaches.

Filters trained in FLAMO or in TorchFX's differentiable module can be imported into Fluxion's freeze pipeline (Sec. III-C). ADAC is a compiler without a filtering runtime of its own, and its published benchmark measures code-generation latency, so we discuss it without measuring it and measure FLAMO, on which it builds. ADAC's loop-aware stability certificate is the closest prior work to our stability verdicts; Fluxion also applies a stability projection during training (Sec. III-B).

Native real-time engines. Faust [1], SuperCollider [2], and JUCE [3] run audio with low latency and complement Fluxion; Faust can also be used for training through its JAX backend [19]. RTNeural [20], the Neutone SDK [21], and anira [22] follow a two-step workflow in which models are trained in Python and then run by a native inference engine. anira is the closest to our setting. It runs ONNX Runtime, LibTorch, and TensorFlow Lite models from an audio callback by moving inference to a thread pool. By default this adds one buffer of latency, and an optional bounded wait in the callback removes it. Fluxion computes the DSP on the audio thread (Sec. IV-C measures both approaches on the same workload) and uses one runtime for training and deployment.

GPU DSP. Parallel IIR filtering on GPUs builds on prefix scans [23]–[25], which TorchFX's CUDA path uses. Fluxion's GPU kernel runs each batch row's cascade sequentially in time, with one GPU thread per row. gpuRIR [26] and Brian Hears [27] vectorise related audio workloads. Burn [12], CubeCL [13], and DLPack [14] are dependencies of Fluxion. Tab. I compares the systems discussed in this section.

III. Architecture

A. Graph algebra and engines

Users describe processing as a functional expression, in the style of Faust's block-diagram algebra [1]. Filters and effects are values, `|` chains them in series, `+` runs them in parallel and sums the outputs, `~` closes a feedback loop, and any node can be named. The expression is stored as a small typed graph with self-describing parameters, so Fluxion can validate it when it is built, serialise it, and lower it to different engines.

TABLE I

FEATURE COMPARISON. ✓ SUPPORTED, ✗ NOT SUPPORTED, ∼ PARTIAL. "ANALYTIC ∇": HAND-DERIVED ADJOINTS INSTEAD OF UNROLLING ON THE HOST TAPE. "PORTABLE GPU": NOT TIED TO ONE VENDOR, WITH VALIDATION STATUS INLINE. "RT VERIF.": AUDIO PATH CHECKED UNDER ALLOCATOR INSTRUMENTATION. "CKPT IMPORT": READS FILTER CHECKPOINTS FROM OTHER FRAMEWORKS.

Framework	GPU	Portable GPU	Analytic ∇	Design-param ∇	Stability cert.	Real-time	RT verif.	Edge footprint	Ckpt import
Fluxion	✓	∼ (CUDA val.)	✓	✓	✓ (+ in-loop)	✓	✓	✓ (O(MB))	✓ (FLAMO, TorchFX)
TorchFX [8]	✓	✗	∼ (opt-in)	∼	✗	∼ (CPU meas.)	✗	✗ (O(GB))	✗
torchaudio [4], [5]	✓	∼ (PyTorch)	∼ (`lfilter`)	∼	✗	✗	✗	✗	✗
GRAFX [7]	✓	∼ (PyTorch)	∼	∼	✗	✗	✗	✗	✗
FLAMO/ADAC [9], [10]	✓	∼ (PyTorch)	✗	✓	∼ (design)	∼ (FAUST exp.)	✗	∼	∼ (FLAMO→FAUST)
anira [22]	✗	—	✗	✗	✗	✓ (default +1 buf.)	∼ (sanitizer)	∼ (O(100 MB))	✓ (ONNX/TS/TFLite)
Faust [1]	✗	—	✗	∼ (JAX)	✗	✓	∼	✓	—
SciPy [28]	✗	—	✗	✗	✗	✗	✗	∼	—

GRAFX cells refer to grafx 0.6.0. torchaudio's `lfilter` has its own differentiable path and its `functional` helpers take cutoff and Q under autograd, hence its two ∼ gradient cells. Fluxion's import cell is tested in CI (Sec. III-C) and covers SISO cascades only.

Fluxion has two engines because training and real-time audio have different requirements. A training step benefits from large batches, free allocation, and often a GPU. An audio callback needs a bounded, deterministic code path, and our earlier measurements showed that per-block GPU dispatch raises throughput but does not lower latency [8]. Fluxion therefore keeps one engine per workload and connects them through a freeze step (Sec. III-C). The sample rate is bound when a graph is lowered, so the same graph can run at any rate.

Both engines use the same traversal: a single walk interprets the graph against a backend trait with one method per kernel, so the backend supplies the buffer type and the kernels and the walk supplies the composition. For training the buffer is a differentiable tensor; for real time, a pre-allocated block of samples. Each lowering rejects the operators it cannot support. The training path rejects operators without an adjoint, and the real-time path rejects operators that need more than the current block, such as peak normalisation. Both currently reject feedback, so the engines cover cascades, FIRs, and delay-line effects.

B. Analytic adjoints

Recursive filters are a poor fit for generic automatic differentiation. If a framework traces the recurrence sample by sample, the tape grows with signal length and cascade depth, and Yu et al. report order-of-magnitude slowdowns for this reason [17]. Fluxion gives each operator a hand-derived adjoint, that is, a closed-form backward pass. A filter section computes

$$y_t = \sum_{i=0}^{M} b_i x_{t-i} - \sum_{j=1}^{N} a_j y_{t-j}. \tag{1}$$

In the backward pass of Eq. (1), the output cotangent goes through a recurrence of the same form, run backwards in time. Input gradients follow by filtering that cotangent with the numerator taps, and coefficient gradients are correlations with the delayed input and output signals. Design-parameter gradients follow by the chain rule through the design map. The design map is closed-form algebra without recursion, so Fluxion computes its Jacobian by central differences, which costs a few extra design evaluations per step and does not grow with signal length.

Fluxion registers these vector-Jacobian products in the autodiff framework the user already works with, as a Burn custom operator or as a custom autodiff operator in PyTorch or JAX. Each operator enters the host graph as one node: forward is the ordinary kernel, backward the hand-derived adjoint, so the host never records the recurrence. The executor that runs a graph on the CPU is the same walk that runs it over autodiff tensors, so series composition chains the adjoints and parallel composition sums the cotangents. Gradients are checked per operator and per composed graph (Sec. IV-B).

Training on design parameters is common in differentiable DSP, where cookbook design equations map cutoff, gain, and Q to coefficients [29], [30]. In Fluxion these parameters sit on top of the analytic adjoint instead of an unrolled tape, so a step costs one forward and one backward pass of the recursion (Tab. III). Within the allowed parameter ranges, every set of design parameters gives a stable filter. When raw coefficients are trained instead, a projection moves each optimiser step back into the stable region (Sec. III-C).

C. Stability check and freezing

Filters are designed in double precision and then cast to the single-precision coefficients that run; we call this cast *freezing*. The stability *certificate* is computed from the frozen values, not from the double-precision design. For an SOS cascade, each section is normalised to $a_0{=}1$ and the largest pole radius after the cast is recorded. Graphs with explicit feedback also get a small-gain check, the maximum loop gain over a linear frequency grid on $[0, \pi]$. The shipped echo and reverb nodes have frequency-flat loop gains, so for them the check is exact; for other loops it is a sampled estimate. The general ˜ operator is checked the same way but is not yet lowered to either engine.

The check returns one of three verdicts: `Stable` (all poles strictly inside the unit circle), `Marginal` (within $\pm 10^{-4}$ of the boundary, accepted but flagged), or `Unstable` (rejected). We chose the band with the cast in mind. Half-ulp rounding of a float32 coefficient is a relative $6{\times}10^{-8}$, and near the unit circle a pole radius moves by about the same amount, so the band is more than three orders of magnitude wider than the rounding error. The frozen graph is stored in a `.fxg` file, a JSON document with a format version and a kind tag, so a

file with an unknown version is rejected instead of misread. A *graph* file stores topology and design parameters and is sample-rate independent; a *frozen-sos* file stores the float32 coefficients and their design rate, so the real-time engine loads it without a design step. The certificate depends only on the file contents, so a device can recompute it at load time instead of trusting the tool that produced the file (Sec. IV-C). It covers stability only and does not authenticate the file.

The same pole and small-gain computations are used at design time, during training, and at export. During training, a projection clamps each section's denominator into the Jury stability triangle shrunk by a margin ($|a_2| \leq 1-m$, $|a_1| \leq 1+a_2-m$, with $m=10^{-3}$ in our experiments; numerators are unchanged) after every optimiser step. The margin is larger than the verdict band because it has to hold until the next step as well as across the cast, and the forward pass only sees iterates inside the stable region. The file contains everything the real-time engine needs, so the trained graph and the graph that runs on the audio thread are the same object (Fig. 3).

Checkpoints trained in FLAMO and in TorchFX's differentiable module can be imported into the same pipeline. The importer replays each library's parameter-to-coefficient map in double precision and applies the same stability check as for a native graph; in CI, fifteen checkpoints from the two libraries reproduce their realised coefficients to within float32 rounding ($< 5\times10^{-5}$ per coefficient). It handles SISO filters only.

D. The real-time engine

The real-time engine runs frozen graphs (cascades, delays, echo, reverb, FIR) block by block with pre-allocated state, and receives parameter changes over a lock-free queue. Preparing a graph for a maximum block size is the only step that allocates: it sizes each node's scratch buffer once, and processing then works in place. The queue is a single-producer, single-consumer ring of plain copyable values, so a coefficient update travels as a fixed-size array of sections instead of a heap-allocated vector. SOS sections run in transposed direct form II on every path, so the certified coefficients describe the same recurrence everywhere. To avoid clicks, new coefficients are designed off the audio thread and faded in with a short linear crossfade. We use a linear fade instead of an equal-power one because both branches process the same input: their outputs are largely correlated, and gains that sum to one approximately keep the level.

Besides measuring latency, we test the audio thread with an instrumented allocator. The test suite installs a global allocator that counts allocations (not frees) inside the audio-thread section. The tests cover a cascade with gain automation sent from a second thread and a nested series–parallel graph, and a control test plants an allocation to confirm that the counter fires. This shows that the tested paths do not allocate, which is weaker than a proof. The queue is lock-free by design, but locks are not instrumented.

GPU execution. The batched forward kernels and the operator interfaces used by their adjoints are written once in a portable kernel language and run directly on tensors kept on the device. We measure this path on CUDA hardware only. The adjoint kernels pass the same gradient checks as the CPU reference in CI; other backends are not yet validated. Sec. IV-A measures when the GPU helps.

Deployment. Fluxion ships as Rust crates with a SoX-style command-line interface, as Python wheels that exchange arrays through DLPack (zero-copy with NumPy, PyTorch, and JAX), and as experimental C-ABI and WebAssembly builds. The stripped release CLI is 3.2 MB on x86-64 and 2.8 MB cross-compiled for ARM64, and it processes 10 s of audio in 20 ms (Tab. IV); a comparable TorchFX deployment first loads CPython and PyTorch.

IV. Evaluation

A. Throughput

1) Setup: We ran all cells of Tab. II on one consumer workstation, an Intel i9-10900KF (20 threads, 32 GB RAM) with an NVIDIA RTX 3070 (8 GB; driver 580.159, CUDA 12.4) under Linux 6.8. The baselines run on Python 3.10.12 with PyTorch 2.10.0+cu128. The edge device of Sec. IV-C is a Raspberry Pi 5 (4×Cortex-A76 at 2.4 GHz, 8 GB, 64-bit Linux 6.18) running Fluxion cross-compiled on the workstation. All measurements use float32 and report the median of five timed repetitions after two warm-up runs (seven after one warm-up for the command-line comparison), and CUDA timings are bracketed by device synchronisation. Multi-thread cells use each library's default thread count, 10 for PyTorch (one per physical core) and 20 for Fluxion; with 10 threads, Fluxion's cells change by -2% to $+13\%$ and no ranking changes. We report GPU results in two regimes: *resident*, with tensors kept on the device as in training, and *one-shot*, where they are uploaded, filtered, and downloaded as in a single utility call. Exact-recurrence systems and frequency-sampling references share the table, but our comparisons refer to the exact-recurrence rows.

Systems compared. Each baseline runs on the path it was designed for. TorchFX [8], our earlier system, is the strongest CPU baseline and has native CUDA kernels. torchaudio's `lfilter` is the exact recurrence used in most PyTorch audio work; it has no SOS primitive, so a cascade costs K kernel dispatches. GRAFX [7] is measured with its exact and its frequency-sampling backends. FLAMO [9] has no time-domain IIR, so its cells report its frequency-sampling computation and are marked as approximate. SciPy's second-order-section routine is the single-threaded C reference, and SoX is the command-line reference. We exclude nnAudio, ADAC [10], and general-purpose inference runtimes because none of them runs a sample-recurrent filter (Sec. II), and compare with anira [22] separately (Sec. IV-C1) on a 1024-tap FIR that both systems can run.

2) Correctness: Before timing, we compared Fluxion's outputs with SciPy. Across the filter families (Butterworth, Chebyshev I/II, the RBJ biquads, FIR), the maximum absolute

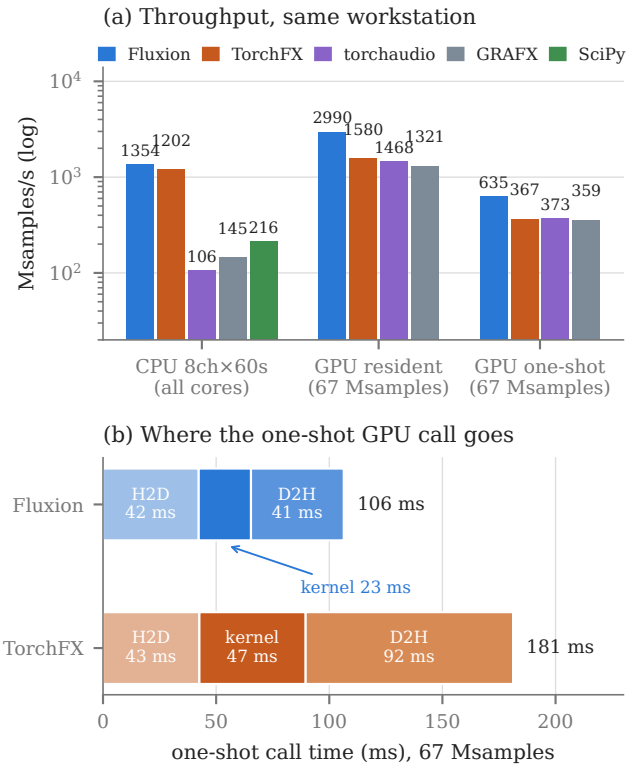

Fig. 2. Throughput on one workstation (RTX 3070 + i9). (a) Cross-system medians, with separate resident and one-shot GPU regimes. (b) One-shot decomposition at 67 Msamples, with device synchronisation between stages. With pinned staging, Fluxion's transfers reach 6.4 GB/s and the kernel takes 23 ms; TorchFX has a slower kernel and a slower download to pageable memory.

deviation from SciPy's reference designs on the same 4000-sample float32 noise signal is 2.7×10^{-5} for the RBJ biquads and at most 5.7×10^{-6} for the other families. A CI test checks that the streaming executor produces the same samples as whole-signal processing.

3) Results: **CPU.** With multiple threads, Fluxion is the fastest exact-recurrence system on the three CPU workloads: 1354 Msamples/s against TorchFX's 1202 on the 8-channel workload, 109 against SciPy's 92 and TorchFX's 63 on the mono 16th-order cascade, and 2603 against 1893 on the 64-signal batch. With one thread, Fluxion is faster on the batch (897 vs. 364) and on the cascade, and TorchFX is faster on the 8-channel workload (407 vs. 288). With multiple threads, Fluxion is 9–41× faster than torchaudio and GRAFX and 1.1–1.7× faster than TorchFX. Its CPU kernels parallelise over the two independent dimensions of an IIR workload [8], sections within a cascade and signals within a batch. The cascade is fused per sample, and batches go through an AVX2 8×8-transpose path with padded scratch tiling to avoid cache conflicts; both paths give bit-identical results to the scalar reference.

FIR. A 2047-tap FIR over 10 s of audio takes 2.7 ms with Fluxion's FFT-based block convolution, 4.8 ms with SciPy's overlap-add convolution, and 44.5 ms with torchaudio's FFT convolution.

TABLE II

CROSS-SYSTEM THROUGHPUT (MSAMPLES/S, MEDIANS, FLOAT32; SINGLE THREAD IN PARENTHESES), CO-RUN ON THE WORKSTATION OF SEC. IV-A1. WORKLOADS: BATCH = 64×524K (3 SECTIONS), 8CH = 8×60 S (ORDER 4), MONO K8 = 1×60 S (ORDER 16); GPU = 16384×4096 (67 MSAMPLES, ORDER 8), RESIDENT / ONE-SHOT (FLUXION: OWNED INPUT BUFFER). †FREQUENCY-SAMPLING APPROXIMATION, NOT THE EXACT RECURRENCE; *SINGLE-THREADED LIBRARY; FLAMO'S BATCH CELL DOES NOT APPLY. BOLD = BEST EXACT-RECURRENCE SYSTEM PER COLUMN (MULTI-THREAD).

System	CPU batch	CPU 8ch	CPU mono K8	GPU res.	GPU 1-shot
Fluxion	**2603** (897)	**1354** (288)	**109** (108)	**2990**	**635**
TorchFX	1893 (364)	1202 (407)	63 (60)	1580	367
torchaudio	64 (20)	106 (32)	9 (8)	1468	373
GRAFX (`lfilter`)	121 (36)	145 (55)	9 (8)	1321	359
GRAFX (`fsm`)†	30 (19)	21 (14)	5 (4)	1199	353
FLAMO†	—	102	14	8964	478
SciPy `sosfilt`*	208	216	92	—	—

Command line. On a SoX task (a 60 s stereo WAV through the same order-16 cascade, from decoding to encoding), the two tools take the same time (Fluxion 176.2 ms, SoX 174.9 ms, medians), and SoX uses less memory (3.8 vs. 5.4 MB peak resident set).

GPU, resident. With tensors on the device, Fluxion's CUDA path processes 2990 Msamples/s, against 1580 for TorchFX's CUDA path, 1468 for torchaudio, and 1321 for GRAFX, and its output matches the CPU executor to within 10^{-4}. On the same workload, Fluxion's multi-threaded CPU path reaches 2662 Msamples/s, so the GPU adds only 12%. Among the approximations, GRAFX's FIR-convolution back-end does not gain from the GPU (1199), while FLAMO's spectral multiplication reaches 8964, much faster than any exact recurrence.

GPU, one-shot. In the one-shot regime, transfers dominate and the kernel takes 23 ms of the round trip. With pinned staging for large uploads, transfers run at 6.4 / 6.5 GB/s and the call takes 106 ms when Fluxion owns the input buffer (Fig. 2b), against about 180 ms for the torch-hosted exact-recurrence systems, which download to pageable memory. With a borrowed slice, as passed by the Python binding, an extra host copy raises the call to 285 ms. At this size the CPU is the better choice for one-shot calls.

B. Differentiable DSP

1) Coloration inversion: A fixed coloration filter, a three-section chain of shelving and peaking filters, colours a signal, and a parametric EQ has to invert it. The trainer sees the coloured and the original signals but not the filter, and minimises the time-domain squared error between the equalised signal and the original. The task is simple, but the ideal inverse is known, so we can measure the residual.

We compare three parameterisations: the EQ's nine design parameters (three frequencies, gains, and Q factors), as proposed in Sec. III-B; the fifteen raw biquad coefficients with the stability projection; and the raw coefficients without

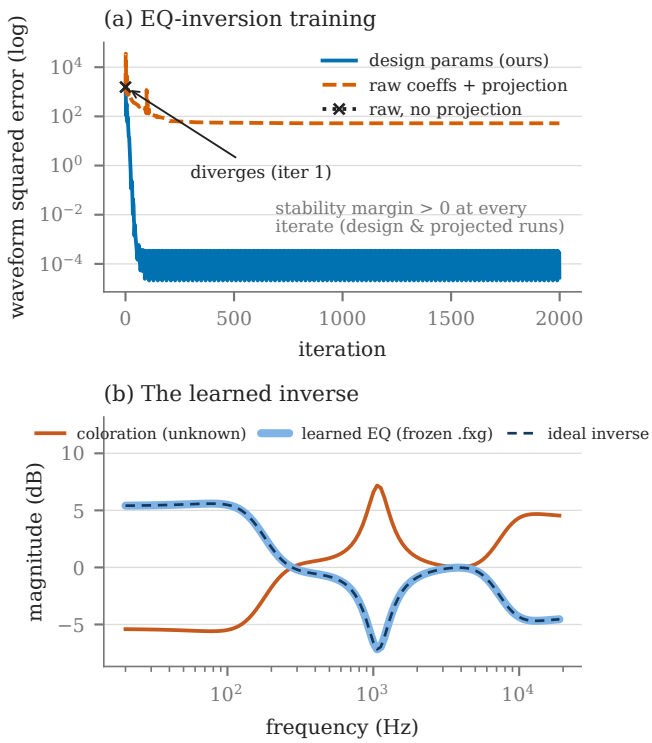

Fig. 3. Coloration inversion with a parametric EQ. (a) Training loss for the three parameterisations. (b) Magnitude response of the frozen artifact trained on design parameters, the file that the real-time engine plays.

projection. Each run takes 2000 iRprop$^-$ steps [31] on the same 8192-sample pink-weighted noise at 48 kHz, using the whole signal at every step. iRprop$^-$ moves each parameter by the sign of its gradient, so we do not need a single learning rate across hertz, decibels, and Q; a step grows by a factor of 1.2 while the sign holds and halves when it flips. Because the optimiser uses only the sign, this task does not test gradient accuracy, which Sec. IV-B3 covers.

With design parameters, the residual is 0.0014 dB RMS (0.004 dB worst case on a 120-point log-frequency grid), and the learned response overlays the ideal inverse in Fig. 3b. The stability margin stays positive by construction. After about 200 iterations the loss stops decreasing and oscillates between 2×10^{-5} and 3.4×10^{-4} (Fig. 3a), an effect of the optimiser's step adaptation. The raw-coefficient run with the projection stays stable at every iterate but stalls at a loss of 52, with a residual of 0.83 dB RMS (2.3 dB worst case); with this optimiser, the nine design parameters are easier to fit than the fifteen coefficients. Without the projection, the first update already leaves the stable region and the forward pass diverges. The EQ trained on design parameters is then frozen and played by the real-time engine, and Fig. 3b shows the response of that file.

Comparison with FLAMO. On the same task, a three-section generic SVF trained as in FLAMO [9] (frequency-sampled magnitude response, dB-MSE, Adam, 4000 iterations in 12.3 s) reaches a residual of 0.44 dB RMS (1.11 dB worst case). This is better than Fluxion's raw-coefficient run and worse than its design-parameter run, so here the parameterisation matters more than the way gradients are computed. The Fluxion runs also give a stability verdict at every iterate and an artifact that the real-time engine plays, whereas FLAMO optimises the magnitude response only and deploys through a separate compiler [10].

TABLE III

ONE TRAINING STEP (FORWARD AND BACKWARD, SQUARED-ERROR LOSS; MS, MEDIAN) OF A K-SECTION CASCADE OVER A BATCH OF 16 SIGNALS OF T SAMPLES. FLUXION: ANALYTIC ADJOINTS ON THE CPU, ONE SIGNAL AT A TIME. TORCHFX `DDSP`: ANALYTIC ADJOINTS OVER THE NATIVE KERNEL (CPU/CUDA). TORCHAUDIO `LFILTER`: ITS BUILT-IN DIFFERENTIABLE PATH (CPU/CUDA). GRAFX: AUTOGRAD OVER ITS EXACT `LFILTER` BACKEND (CPU). FLAMO†: FREQUENCY-SAMPLING STEP (MAGNITUDE-RESPONSE MSE AT $nfft{=}T$), WHOSE COST BARELY DEPENDS ON K. AN UNROLLED RECURRENCE TAKES 408 MS AT $K{=}1, T{=}4$K, ABOUT 153× THE ANALYTIC PATH PER SAMPLE.

K	T	Fluxion	TorchFX ddsp	torchaudio	GRAFX	FLAMO†
1	16k	11	10 / 15	4 / 7	3	3
1	64k	46	22 / 16	17 / 30	—	—
4	16k	26	38 / 63	15 / 26	13	5
4	64k	114	82 / 63	75 / 113	—	—
8	16k	48	73 / 126	31 / 52	26	5
8	64k	207	162 / 126	149 / 221	—	—

2) Training cost: Tab. III reports one forward and backward step of an SOS cascade with five methods: Fluxion's analytic adjoints, TorchFX's differentiable module, torchaudio's built-in differentiable recursion, GRAFX's autograd over the same backend, and FLAMO's frequency-sampling step. FLAMO's step computes one FFT-domain multiplication per section, so its cost barely depends on K (3–5 ms), and its gradient approximates the exact recursion that the other rows compute. The Fluxion column uses the CPU adjoint path and processes the signals of the batch one at a time. On CPU it is slower than torchaudio and GRAFX in every measured cell (1.4–3.2×), and faster than TorchFX only at $T{=}16$k with $K \geq 4$. The caption gives the cost of an unrolled recursion.

GPU training. The analytic VJP kernels also run on the GPU with data kept on the device, in two configurations. Gradients through a frozen cascade to its input, as needed for a DSP layer inside a larger model, cost 40–144 ms per step at $T{=}16$k and 137–714 ms at $T{=}64$k for $K \in \{1, 4, 8\}$ (batch 16). A trainable single section costs 32 ms at $T{=}16$k and 116 ms at $T{=}64$k, which is slower than Fluxion's CPU adjoint (11 and 46 ms) because the current entry point launches one row at a time and synchronises the device after each row. A batched trainable cascade is future work. For now, the CPU path trains faster at batch 16, and the GPU helps only in the resident forward path of Sec. IV-A.

3) Gradient checks: Finite differences are unreliable through a deep recursive filter, especially in single precision, so we check gradients at two levels. Each operator's VJP (input, coefficient, design-parameter, FIR, and delay gradients) is compared with finite differences in isolation, where the comparison is well conditioned. The gradient of a mixed

TABLE IV
DEPLOYMENT FOOTPRINT. THE FLUXION ROWS PROCESS 10 S OF AUDIO (16TH-ORDER CASCADE AND REVERB, FILE TO FILE) ON THE WORKSTATION AND ON A RASPBERRY PI 5 (BINARY CROSS-COMPILED, ZIG-LINKED); THE PYTHON ROW IS THE COST OF `import torch`, `torchfx` ALONE.

Stack	On disk	Peak RSS	Wall
Fluxion CLI (x86-64, stripped)	3.2 MB	11 MB	20 ms
Fluxion CLI (Pi 5, aarch64)	2.8 MB	11 MB	46 ms
CPython + torch + torchfx (import only)	1.7 GB	625 MB	4.3 s

series–parallel graph is compared with an independently hand-composed adjoint to within 10^{-4}, and the GPU backward kernels pass the same checks on the device. All checks run in continuous integration.

C. Real-Time and Edge

1) Real-time behaviour: The allocation tests of Sec. III-D run in CI and find no allocations on the tested audio-thread paths. Fig. 4 shows the per-block processing time at cascade depths $d \in \{2, 5, 10\}$ and buffer sizes $\{128, 256, 512\}$ (deadlines of 2.7–10.7 ms at 48 kHz), with coefficient crossfades running throughout. Blocks are processed one after another, without pacing or real-time priority, so the figure shows computation cost and not operating-system scheduling delays.

In every configuration, the p99 is at most 0.26% of the deadline (6.8 μs at depth 10 with 128-sample buffers). The slowest of the 73,728 blocks took 872 μs (depth 5, 512 samples, 8% of that deadline); in the other configurations, the slowest block took less than 66 μs. Our earlier study measured a TorchFX p99 of 0.71–0.74% of the deadline on the same machine at 256 samples, the smallest buffer it used.

Comparison with anira. anira [22] runs neural inference outside the audio callback, on a thread pool. By default the callback returns an earlier result, which adds one buffer of latency (2.7–10.7 ms here). With a blocking ratio $W > 0$, the callback waits up to W buffer periods for the current result, and anira adds no latency when that wait covers the declared worst-case inference time. We ran a 1024-tap FIR with the same taps in both systems: anira runs it as an ONNX convolution (ONNX Runtime, W=0.75, which adds no latency at the three buffer sizes), and Fluxion runs its real-time FIR node. Both run unpaced and without real-time priority, over at least 5,000 blocks per configuration. At 128/256/512 samples, Fluxion processes a block in 25/51/101 μs (median; p99 27/68/124 μs), and anira's callback returns in 81/91/150 μs (p99 173/177/227 μs); no anira block reached the wait limit. The anira process has a 363 MB resident set and the Fluxion process 2.7 MB. anira can run arbitrary neural models and Fluxion cannot, so this comparison applies only to DSP workloads.

2) Footprint: Tab. IV compares the two stacks on the same task. Fluxion's ARM64 release binary is a 2.8 MB executable without Python or PyTorch dependencies, while a TorchFX deployment loads CPython and PyTorch before it processes the first sample.

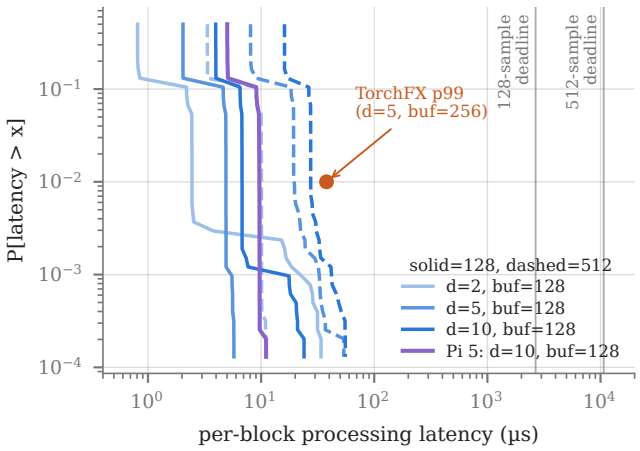

Fig. 4. Per-block processing time (unpaced) at 128 and 512 samples. The violet curve is the same code on a Raspberry Pi 5; the orange point is TorchFX's p99 from our earlier study.

Raspberry Pi 5. The cross-compiled binary runs the 16th-order cascade and reverb chain at 217× real time with an 11 MB resident set on a Pi 5 (4×Cortex-A76, 8 GB). With the unpaced protocol above and coefficient crossfades running, the p99 is at most 0.36% of the deadline over all depths and buffer sizes, and the slowest block took 41 μs (Fig. 4).

Hot-swap on the device. We ran the train, certify, push, and crossfade sequence of Sec. V-B end to end. The EQ of Sec. IV-B, trained on the workstation, was copied as a frozen `.fxg` file over the LAN to the Pi 5 while the engine streamed 128-sample blocks. The device recomputes the certificate when the file arrives and then crossfades the new filter in through the lock-free command queue. Loading, certification, freezing, and enqueueing took 27.5 μs on the device. The whole update, from the start of the copy to the engine's acknowledgement, took 1.24 s, mostly for the network copy and polling. During the 43 ms crossfade, the largest sample-to-sample step of the output was 0.036, against 0.029 in steady state. In the same run, a file edited after certification to make it unstable (echo feedback changed from 0.9 to 1.5) was rejected on the device and never reached the audio thread.

3) Case study: SoundFood: SoundFood (https://www.soundfood.it/) develops a smart speaker for restaurants (Fig. 5) that plays AI-generated music matched to each dish [32], [33], an application of "sonic seasoning", the use of sound to shape taste perception [34]. Because tables start their dishes at different times, the device keeps each table's music within a local "sound bubble". The production chain upmixes a mono track to eight channels (7.1). Each channel runs six frozen biquad sections (two order-4 Butterworth band limits, one shelving and one peaking section), its own 1024-tap directional FIR, and a level trim, and a separate low-frequency feed uses an order-2 Linkwitz-Riley crossover and an order-4 Butterworth. We report the structure of the chain; its tuning (cutoffs, gains, and taps) belongs to SoundFood.

Fig. 5. The SoundFood soundlamp, a 7.1 smart speaker that plays AI-generated music matched to each dish in restaurants. Its equalisation and directional FIR chain runs in real time on a Raspberry Pi-class core (Sec. IV-C3).

The system started as an offline batch service (FastAPI over TorchFX) that pre-rendered and stored every track eight times (5.3 MB of mono source becomes 42.3 MB per minute of 7.1 output), holding the whole file in memory (0.6 GB peak for a one-minute track). On the device, the newest TorchFX wheel for ARM runs the chain at $0.72\times$ real time (83.8 s for a 60 s track, 661 MB peak, a 968 MB installation, and a 2.1 s import), and TorchFX's compiled kernels are built for x86 only.

Fluxion loads the frozen biquad rows and FIR taps of the production chain directly. The largest output deviation from the production service is 2.0×10^{-4} full scale on one channel and at most 9.6×10^{-5} on the others; the reference output is 16-bit PCM (step 3.1×10^{-5}). On the Pi 5, Fluxion runs the chain as eight separate certified graphs. Paced at the audio rate, the p99 block latency is at most 33% of the deadline at buffer sizes of 128, 512, and 1024 samples, with no missed deadlines over 57,000 blocks, 31% of one core, and a 13.5 MB resident set from a 468 KB harness binary. Offline, the chain runs at $4.2\times$ real time on the device. The equalisation can therefore run on the device instead of being pre-rendered, and per-dish retuning could use the hot-swap described above.

V. Discussion

A. Summary of results

Among the exact-recurrence systems, Fluxion has the highest throughput in the resident GPU regime and in most CPU cells (with one thread, TorchFX is faster on the 8-channel workload), and it takes the same time as SoX on the file-to-file task. Its CPU training step is slower than those of torchaudio and GRAFX, and its GPU training path is still slower than its CPU path. The deployment footprint is about $500\times$ smaller on disk and $57\times$ smaller in memory than the Python stack, and none of the measured baselines covers training, certification, freezing, and playback in one library. Fluxion is a DSP runtime and does not offer torchaudio's range of transforms.

B. Research directions

Design-parameter gradients turn filter design into an optimisation over interpretable parameters that keep the filter stable. This could be used to study perceptual losses, style transfer between EQ curves, and regularisation that is awkward to express over raw coefficients. Versioned frozen artifacts, a stability check at export, and crossfaded coefficient swaps let edge devices replace retrained DSP while streaming, with stability checked on the artifact itself. Managing fleets of devices, versioning, and staged rollout of DSP updates are open problems.

A frequency-sampling step like FLAMO's, whose cost barely depends on depth (Tab. III), could be added to Fluxion for coarse training, followed by certification and fine-tuning with the exact adjoints on the same graph. An adjoint for the feedback operator, together with a loop-aware stability check during training, would make learnable reverberators and feedback delay networks possible; keeping these stable during training is hard, and the stability verdicts of Sec. III-C could be a starting point.

C. Limitations

GPU execution is validated on CUDA only. The kernel layer also targets Metal, ROCm, and WebGPU, but we have not tested it on that hardware. GPU training is measured only for the single trainable section and for gradients through frozen cascades (Sec. IV-B), and trainable multi-section cascades run on the CPU adjoint path. The feedback operator executes but is neither differentiable nor lowered to the real-time engine. The kernels do not force flush-to-zero for denormals, which matters for decaying tails near digital silence. Fluxion has no neural operators of its own. The allocation claim relies on instrumented tests of specific paths, not on static analysis of all reachable paths. Frees and locks are not instrumented, and the property holds per process, not at the operating-system level.

D. Threats to validity

All numbers come from one workstation and one edge board, and other hardware could change the order of close cells (the smallest CPU margin is 13%). Each baseline runs on its own best path under one float32 protocol, and we state the remaining protocol differences where the numbers appear: frequency-sampling approximations, SoX's file-to-file pipeline, the callback-to-return measurement of our earlier study, the FIR-only comparison with anira, and the default thread counts. Timings are medians of five repetitions after warm-up, and Tab. II and Tab. III report them without dispersion, so the closest comparisons (the 13% CPU cell and the command-line comparison) should be read as ties. The latency distributions of Sec. IV-C were measured without pacing or real-time priority.

VI. Conclusions and Future Work

Fluxion implements the DSP-specific numerical code (closed-form design, forward kernels, analytic adjoints, and stability checks) and uses existing libraries for autodiff, GPU code generation, and array interchange. One graph algebra is lowered to a training engine and to a real-time engine whose audio thread we test for allocations, and the freeze step carries

a checked artifact from training to the edge device, including checkpoints imported from FLAMO and TorchFX.

Future work includes testing the Metal, ROCm, and WebGPU kernels on hardware, making the feedback operator differentiable and usable in real time with a loop-aware stability check during training, adding frequency-sampling gradients as a training option, and defining versioning and rollout policies for DSP updates on many devices. We do not plan fp16 or bf16 execution, because recursive filters are sensitive to coefficient quantisation at half precision.

Disclosure of AI Use

We used large language model assistants when drafting and revising parts of this manuscript and as coding assistants while implementing Fluxion. They were not used to produce or to interpret any measurement. The authors derived and checked the adjoints, designed and ran every experiment, verified the code and the numbers reported here, and take full responsibility for them.

Acknowledgment

This work was funded by the European Union - NextGenerationEU, under the National Recovery and Resilience Plan (PNRR).

References

[1] Y. Orlarey, D. Fober, and S. Letz, "FAUST : an Efficient Functional Approach to DSP Programming," in *NEW COMPUTATIONAL PARADIGMS FOR COMPUTER MUSIC*, E. D. FRANCE, Ed., 2009, pp. 65–96. [Online]. Available: https://hal.science/hal-02159014

[2] J. McCartney, "Rethinking the computer music language: Supercollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 12 2002.

[3] Raw Material Software Limited, "JUCE: Cross-platform C++ framework for audio applications," https://juce.com, 2026, accessed: 2026-05-29.

[4] Y.-Y. Yang, M. Hira, Z. Ni, A. Astafurov, C. Chen, C. Puhrsch, D. Pollack, D. Genzel, D. Greenberg, E. Z. Yang, J. Lian, J. Hwang, J. Chen, P. Goldsborough, S. Narenthiran, S. Watanabe, S. Chintala, and V. Quenneville-Bélair, "Torchaudio: Building blocks for audio and speech processing," in *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2022, pp. 6982–6986.

[5] J. Hwang, M. Hira, C. Chen, X. Zhang, Z. Ni, G. Sun, P. Ma, R. Huang, V. Pratap, Y. Zhang, A. Kumar, C.-Y. Yu, C. Zhu, C. Liu, J. Kahn, M. Ravanelli, P. Sun, S. Watanabe, Y. Shi, and Y. Tao, "Torchaudio 2.1: Advancing speech recognition, self-supervised learning, and audio processing components for pytorch," in *2023 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*, Dec 2023, pp. 1–9.

[6] K. W. Cheuk, H. Anderson, K. Agres, and D. Herremans, "nnaudio: An on-the-fly gpu audio to spectrogram conversion toolbox using 1d convolutional neural networks," *IEEE Access*, vol. 8, pp. 161 981–162 003, 2020.

[7] S. Lee, M. A. Martínez-Ramírez, W.-H. Liao, S. Uhlich, G. Fabbro, K. Lee, and Y. Mitsufuji, "Grafx: An open-source library for audio processing graphs in pytorch," in *Proceedings of the International Conference on Digital Audio Effects, DAFx*, B. S., R. J.D., N. A., and D. S. E., Eds. DAFx, 2024, p. 475 – 478.

[8] M. Spanio and A. Rodà, "Torchfx: A modern approach to audio dsp with pytorch and gpu acceleration," in *Proceedings of the International Conference on Digital Audio Effects, DAFx*. DAFx, 2025, p. 390 – 395.

[9] G. Dal Santo, G. M. De Bortoli, K. Prawda, S. J. Schlecht, and V. Välimäki, "Flamo: An open-source library for frequency-domain differentiable audio processing," in *ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, April 2025, pp. 1–5.

[10] F. Franchino and S. J. Schlecht, "Compiling differentiable audio graphs to real-time dsp," in *Proceedings of the International Conference on Digital Audio Effects, DAFx*, 2026.

[11] L. Turchet, M. Lagrange, C. Rottondi, G. Fazekas, N. Peters, J. Østergaard, F. Font, T. Bäckström, and C. Fischione, "The Internet of Sounds: Convergent trends, insights, and future directions," *IEEE Internet of Things Journal*, vol. 10, no. 13, pp. 11 264–11 292, Jul. 2023.

[12] Tracel AI, "Burn: A next generation deep learning framework," 2026. [Online]. Available: https://burn.dev

[13] N. Simard, L. Fortier-Dubois, G. Lagrange, G. Wingert, A. Brussee, and CubeCL Contributors, "CubeCL," May 2026, version 0.10.0. [Online]. Available: https://github.com/tracel-ai/cubecl

[14] DMLC, "DLPack: Open in-memory tensor structure," 2023. [Online]. Available: https://github.com/dmlc/dlpack

[15] J. Engel, L. H. Hantrakul, C. Gu, and A. Roberts, "Ddsp: Differentiable digital signal processing," in *International Conference on Learning Representations*, 2020.

[16] M. Forgione and D. Piga, "dynoNet: A neural network architecture for learning dynamical systems," *International Journal of Adaptive Control and Signal Processing*, vol. 35, no. 4, pp. 612–626, 2021.

[17] C.-Y. Yu, C. Mitcheltree, A. Carson, S. Bilbao, J. D. Reiss, and G. Fazekas, "Differentiable all-pole filters for time-varying audio systems," in *Proceedings of the International Conference on Digital Audio Effects, DAFx*, B. S., R. J.D., N. A., and D. S. E., Eds. DAFx, 2024, Conference paper, p. 345 – 352.

[18] C.-Y. Yu and G. Fazekas, "Accelerating automatic differentiation of direct form digital filters," in *1st Workshop on Differentiable Systems and Scientific Machine Learning @ EurIPS*, 2025, arXiv:2511.14390.

[19] D. Braun, "DawDreamer: Digital audio workstation with Python; VST instruments/effects, parameter automation, FAUST, JAX, warp markers, and JUCE processors," https://github.com/DBraun/DawDreamer, 2026, accessed: 2026-09-13.

[20] J. Chowdhury, "Rtneural: Fast neural inferencing for real-time systems," *arXiv preprint arXiv:2106.03037*, 2021.

[21] C. Mitcheltree, B. Teleaga, A. Fyfe, N. Masuda, M. Schäfer, A. Bradic, and N. Tokui, "Neutone SDK: An open source framework for neural audio processing," in *AES International Conference on Artificial Intelligence and Machine Learning for Audio*, 2025.

[22] V. Ackva and F. Schulz, "Anira: An architecture for neural network inference in real-time audio applications," in *2024 IEEE 5th International Symposium on the Internet of Sounds (IS2)*, Sep. 2024, pp. 1–10.

[23] G. E. Blelloch, "Prefix sums and their applications," School of Computer Science, Carnegie Mellon University, Tech. Rep. CMU-CS-90-190, Nov. 1990.

[24] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan Primitives for GPU Computing," in *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, M. Segal and T. Aila, Eds. The Eurographics Association, 2007.

[25] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Addison-Wesley Professional, 2007, ch. 39, pp. 851–876.

[26] D. Diaz-Guerra, A. Miguel, and J. R. Beltran, "gpurir: A python library for room impulse response simulation with gpu acceleration," *Multimedia Tools and Applications*, vol. 80, no. 4, pp. 5653–5671, 2021.

[27] B. Fontaine, D. F. M. Goodman, V. Benichoux, and R. Brette, "Brian hears: Online auditory processing using vectorization over channels," *Frontiers in Neuroinformatics*, vol. Volume 5 - 2011, 2011.

[28] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau *et al.*, "Scipy 1.0: Fundamental algorithms for scientific computing in python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, 2020.

[29] S. Nercessian, "Neural parametric equalizer matching using differentiable biquads," in *Proc. 23rd Int. Conf. Digital Audio Effects (DAFx-20)*, Vienna, Austria, 2020, pp. 265–272.

[30] C. J. Steinmetz, J. Pons, S. Pascual, and J. Serrà, "Automatic multitrack mixing with a differentiable mixing console of neural audio effects," in *ICASSP 2021 – 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2021, pp. 71–75.

[31] C. Igel and M. Hüsken, "Improving the Rprop learning algorithm," in *Proc. Second International ICSC Symposium on Neural Computation (NC 2000)*. ICSC Academic Press, 2000, pp. 115–121.

[32] M. Spanio, "Towards emotionally aware ai: Challenges and opportunities in the evolution of multimodal generative models," in *Proceedings of the AIxIA Doctoral Consortium 2024 co-located with the 23nd International*

Conference of the Italian Association for Artificial Intelligence (AIxIA 2024), ser. CEUR Workshop Proceedings, vol. 3914, 2024.

[33] M. Spanio, M. Zampini, A. Rodà, and F. Pierucci, "A multimodal symphony: integrating taste and sound through generative AI," *Frontiers in Computer Science*, vol. 7, 2025.

[34] M. Auvray and C. Spence, "The multisensory perception of flavor," *Consciousness and Cognition*, vol. 17, no. 3, pp. 1016–1031, 2008.