A. RELATED WORKS

A.1. Composition of Audio Processors

Most audio processors are designed to modify some specific properties of their input signals, e.g., magnitude response, loudness dynamics, and stereo width. As such, combining multiple processors is a common practice to achieve the full effect. Following the main text, we will use the terminology "graph" to represent this composition, although some previous works considered simple structures that allow a more compact form, e.g., a sequence. Now, we outline the previous attempts that tried to estimate the processing graph or its parameters from reference audio. These works differ in task and domain, processors, considered graph structures, and estimation methods. For example, if the references are dry sources and a wet mixture, this task becomes reverse engineering [1, 2, 3, 4]. In terms of the prediction targets, some fixed the graph and estimated only the parameters [5, 6, 4, 7, 8]. Others tried to predict the graph [3] or both [1, 2, 9, 10]. Such differences are summarized and highlighted in Table 1.

A.2. Differentiable Signal Processing

Differentiable processor — Exact implementation or approximation of processors in an automatic differentiation framework, e.g., `pytorch` [11], enables parameter optimization via gradient descent. Numerous efforts have been focused on converting existing audio processors to their differentiable versions [4, 12, 13, 14, 15, 16]; refer to the recent review [17] and references therein for more details. In many cases, these processors are combined with neural networks, whose computation is done in GPU. Thus, converting the audio processors to be "GPU-friendly" has been an active research topic. For example, for a linear time-invariant (LTI) system with a recurrent structure, we can sample its frequency response to approximate its infinite impulse response (IIR) instead of directly running the recurrence; the former is faster than the latter [15, 16]. However, it is nontrivial to apply a similar trick to nonlinear, recurrent, or time-varying processors. Typically, further simplifications and approximations are employed, e.g., replacing the nonlinear recurrent part with an IIR filter [12] or assuming frame-wise LTI to a linear time-varying system [14]. Sometimes, we can only access input and output signals. In such a case, one can approximate the gradients with finite difference methods [8, 18] or use a pretrained auxiliary neural network that mimics the processors [5]. In the literature, these are also referred to as "differentiable;" hence, it is rather an umbrella term encompassing all methods that obtain the output signals or gradients within a reasonable amount of time. Nevertheless, our work limits the focus to the implementations in the automatic differentiation framework.

Audio processing graph — Now, consider a composition of multiple differentiable processors; the entire graph remains differentiable due to the chain rule. However, the following practical considerations remain. If we fix the processing graph prior to the optimization and the graph size is relatively small, we can implement the "differentiable graph" following the existing implementations [7, 6]. That is, we compute every processor one by one in a pre-defined topological order. However, we have the following additional requirements. First, the pruning changes the graph during the optimization. Therefore, our implementation must take a graph and its parameters along with the source signals as input arguments for every forward pass. Note that this feature is also necessary when training a neural network that predicts the parameters of any given graph [1]. Second, the size of our graphs is much larger than the ones from previous works [7, 6, 9, 1]. In this case, the one-by-one computation severely bottlenecks the computation speed. Therefore, we derived a flexible and efficient graph computation algorithm (i) that can take different graphs for each forward pass as input and (ii) performs batched processing of multiple processors within a graph, utilizing the parallelism of GPUs. Finally, we note that other than the differentiation with respect to the input signals $\mathbf{S}$ and parameters $\mathbf{P}$, one might be interested in differentiation with respect to the graph structure G. The proposed pruning method performs this to a limited extent; deletion of a node v_i is a binary operation that modifies the graph structure. We relaxed this to a continuous dry/wet weight w_i and optimized it with the audio loss L_{a} and regularization L_{p}. We leave differentiable relaxation of other operations as future work (also see the following section).

A.3. Graph Search

Several independent research efforts in various domains exist that search for graphs that satisfy certain requirements. For example, neural architecture search (NAS) aims to find a neural network architecture that achieves an improved performance [19]. In this case, the search space consists of graphs, each node (or edge) representing one of the primitive neural network layers. One particularly relevant work to ours is a differentiable architecture search (DARTS) [20], which relaxes the choice of each layer to a categorical distribution and optimizes it via gradient descent. Theoretically, our method can be naturally extended to this approach; we only need to change our 2-way choice (prune or not) to $(N+1)$-way (bypass or select one of N processor types). DARTS is clearly more flexible and general as it allows arbitrary order of processors. However, at the same time, it greatly increases the computational cost as we must compute all N processors to compute their weight sum for every node. For example, if we want to keep the mixing console structure and allow arbitrary choices of the processor, the memory complexity becomes $O(N^2)$ instead of the current $O(N)$. In other words, we must pay additional costs to increase the size of the search space. This cost increase is especially critical to us since we have to find a graph for every song. Another popular related domain is the generation/design of molecules with desired chemical properties [21]. One dominant approach for this task is to use reinforcement learning (RL), which estimates each graph by making a sequence of decisions, e.g., adding nodes and edges [22]. RL is an attractive choice since we can be completely free with prior assumptions on graphs, and we can use arbitrary quality measures that are not differentiable. We also note that RL can be used for NAS [23]. However, applying RL to our task has a risk of obtaining nontrivial mixing graphs that are difficult for practitioners to interpret; we may need a soft regularization penalty that guides the generation process towards familiar structures, e.g., ones like the pruned mixing consoles. Also, it may need much larger computational resources to explore the search space sufficiently.

B. DIFFERENTIABLE AUDIO PROCESSORS

For each processor, $u[n]$ and $y[n]$ denote its stereo input and output, respectively. We write the discrete-time index with n. p with a subscript denotes a real-valued parameter vector, which could be a concatenation of multiple parameters. Each subscript x denotes one of the following channels: left l, right r, mid m, and side s.

[6]	*Task & domain*	Sound matching $[x] \rightarrow [\mathbf{P}]$. The synthesizer parameters $\mathbf{P}$ were estimated to match the reference (target) audio x.
	Processors	Oscillators, envelope generators, and filters that allow parameter modulation as an optional input.
	Graph	Any pre-defined directed acyclic graph (DAG). For example, a subtractive synthesizer that comprises 2 oscillators, 1 amplitude envelope, and 1 lowpass filter were used in the experiments.
	Method	Trained a single neural backbone for the reference encoding, followed by multiple prediction heads for the parameters. Optimized with a parameter loss and spectral loss, where the latter is calculated with every intermediate output.
[10]	*Task & domain*	Sound matching $[x] \rightarrow [\mathbf{P}]$. A frequency-modulation (FM) synthesizer matches recordings of monophonic instruments (violin, flute, and trumpet). Estimates parameters of an operator graph that is empirically searched & selected.
	Processors	Differentiable sinusoidal oscillators, each used as a carrier or modulator, pre-defined frequencies. An additional FIR reverb is added to the FM graph output for post-processing.
	Graph	DAGs with at most 6 operators. Different graphs for different target instruments.
	Method	Trained a convolutional neural network that estimates envelopes from the target loudness and pitch.
[9]	*Task & domain*	Sound matching $[x] \rightarrow [G, \mathbf{P}]$. Similar setup to the above [10] plus additional estimation of the operator graph G.
	Processors	Identical to [10], except for the frequency ratio that can be searched.
	Graph	A subgraph of a supergraph, which resembles a multi-layer perceptron (modulator layers followed by a carrier layer).
	Method	Trained a parameter estimator for the supergraph and found the appropriate subgraph G with an evolutionary search.
[2]	*Task & domain*	Reverse engineering $[s, y] \rightarrow [G, \mathbf{P}]$ of an audio effect chain from a subtractive synthesizer (commercial plugin).
	Processors	5 audio effects: compressor, distortion, equalizer, phaser, and reverb. Non-differentiable implementations.
	Graph	Chain of audio effects generated with no duplicate types (therefore 32 possible combinations) and random order.
	Method	Trained a next effect predictor and parameter estimator in a supervised (teacher-forcing) manner.
[3]	*Task & domain*	Blind estimation $[y] \rightarrow [G]$ and reverse engineering $[s, y] \rightarrow [G]$ of guitar effect chains.
	Processors	13 guitar effects, including non-linear processors, modulation effects, ambience effects, and equalizer filters.
	Graph	A chain of guitar effects. Maximum 5 processors and a total of 221 possible combinations.
	Method	Trained a convolutional neural network with synthetic data to predict the correct combination.
[5]	*Task & domain*	Automatic mixing $[\mathbf{S}] \rightarrow [\mathbf{P}]$. Estimated parameters of fixed processing chains from source tracks ($K \leq 16$).
	Processors	7 differentiable processors, where 4 (gain, polarity, fader, and panning) were implemented exactly. A combined effect of the remaining 3 (equalizer, compressor, and reverb) was approximated with a single pre-trained neural network.
	Graph	Tree structure: applied a fixed chain of the 7 processors for each track, and then summed the chain outputs altogether.
	Method	Trained a parameter estimator (convolutional neural network) with a spectrogram loss end-to-end.
[8]	*Task & domain*	Reverse engineering of music mastering $[s, y] \rightarrow [\mathbf{P}]$
	Processors	A multi-band compressor, graphic equalizer, and limiter. Gradient approximated with a finite difference method.
	Graph	A serial chain of the processors.
	Method	Optimized parameters with gradient descent.
[1]	*Task & domain*	Blind estimation $[y] \rightarrow [G, \mathbf{P}]$ and reverse engineering $[\mathbf{S}, y] \rightarrow [G, \mathbf{P}]$. Estimates both graph and its parameters for singing voice effect ($K = 1$) or drum mixing ($K \leq 6$).
	Processors	A total of 33 processors, including linear filters, nonlinear filters, and control signal generators. Some processors are multiple-input multiple-output (MIMO), e.g., allowing auxiliary modulations. Non-differentiable implementations.
	Graph	Complex DAG; splits (e.g., multi-band processing) and merges (e.g., sum and modulation). 30 processors max.
	Method	Trained a convolutional neural network-based reference encoder and a transformer variant for graph decoding and parameter estimation. Both were jointly trained via direct supervision of synthetic graphs (e.g., parameter loss).
[4]	*Task & domain*	Reverse engineering $[\mathbf{S}, y] \rightarrow [\mathbf{P}]$ of music mixing. Estimated parameters of a fixed chain for each track.
	Processors	6 differentiable processors: gain, equalizer, compressor, distortion, panning, and reverb.
	Graph	A chain of 5 processors (all above types except the reverb) for each dry track (any other DAG can also be used). The reverb is used for the mixed sum.
	Method	Parameters were optimized with spectrogram loss end-to-end via gradient descent.
Ours	*Task & domain*	Reverse engineering $[\mathbf{S}, y] \rightarrow [G, \mathbf{P}]$ of music mixing. Estimated a chain of processors and their parameters for each track and submix where $K \leq 130$.
	Processors	7 differentiable processors: gain/panning, stereo imager, equalizer, reverb, compressor, noisegate, and delay.
	Graph	A tree of processing chains with a subgrouping structure (any other DAG can also be used). Processors can be omitted but should follow the fixed order.
	Method	Joint estimation of the soft masks (dry/wet weights) and processor parameters. Optimized with the spectrogram loss (and additional regularizations) end-to-end via gradient descent. Accompanied by hard pruning stages.

Table 1: A brief summary and comparison of previous works on estimation of compositional audio signal processing.

B.1. Gain/panning

We use simple channel-wise constant multiplication. Its parameter vector $p_{\mathrm{g}} \in \mathbb{R}^2$ is in log scale, so we apply exponentiation before multiplying it to the stereo signal.

$$y[n] = \exp(p_{\mathrm{g}}) \cdot u[n]. \tag{1}$$

B.2. Stereo Imager

We multiply the side signal (left minus right) with a gain parameter $p_{\mathrm{s}} \in \mathbb{R}$ to control the stereo width. The mid and side outputs are

$$y_{\mathrm{m}}[n] = u_{\mathrm{l}}[n] + u_{\mathrm{r}}[n], \tag{2a}$$
$$y_{\mathrm{s}}[n] = \exp(p_{\mathrm{s}}) \cdot (u_{\mathrm{l}}[n] - u_{\mathrm{r}}[n]). \tag{2b}$$

Then, we convert the mid/side output to a stereo signal as follows, $y_{\mathrm{l}}[n] = (y_{\mathrm{m}}[n] + y_{\mathrm{s}}[n])/2$ and $y_{\mathrm{r}}[n] = (y_{\mathrm{m}}[n] - y_{\mathrm{s}}[n])/2$.

B.3. Equalizer

We employ a zero-phase FIR filter. Considering its log-magnitude as a parameter p_{e}, we compute inverse FFT (IFFT) of the magnitude response and multiply it with a Hann window $v^{\mathrm{Hann}}[n]$. As a result, the length-N FIR is given as

$$h_{\mathrm{e}}[n] = v^{\mathrm{Hann}}[n] \cdot \frac{1}{N} \sum_{k=0}^{N-1} \exp p_{\mathrm{e}}[k] \cdot w_N^{kn} \tag{3}$$

where $-(N+1)/2 \leq n \leq (N+1)/2$ and $w_N = \exp(j \cdot 2\pi/N)$. We compute the final output by applying the same FIR to both the left and right channels as follows,

$$y_{\mathrm{x}}[n] = u_{\mathrm{x}}[n] * h_{\mathrm{e}}[n] \quad (\mathrm{x} \in \{\mathrm{l}, \mathrm{r}\}). \tag{4}$$

We set the FIR length to $N = 2047$. Therefore, the parameter p_{e} has a size of 1024. Note that we could use a parametric equalizer [15] as an alternative for more compact parameters. However, we used this FIR filter due to its simplicity and fast computation.

B.4. Reverb

We use a variant of filtered noise models [24, 7]. First, we create 2 seconds of uniform noises, $u_{\mathrm{m}}[n]$ and $u_{\mathrm{s}}[n]$. Next, we multiply each noise's STFT $U_{\mathrm{x}}[k, m]$ with a magnitude mask $M_{\mathrm{x}}[k, m]$.

$$H_{\mathrm{x}}[k, m] = U_{\mathrm{x}}[k, m] \odot M_{\mathrm{x}}[k, m] \quad (\mathrm{x} \in \{\mathrm{m}, \mathrm{s}\}). \tag{5}$$

Here, k and m denote frequency and time frame index. Each mask is parameterized with an initial log-magnitude $H_{\mathrm{x}}^0[k]$ and an absorption filter $H_{\mathrm{x}}^{\Delta}[k]$ as follows,

$$M_{\mathrm{x}}[k, m] = \exp(H_{\mathrm{x}}^0[k] + (m-1)H_{\mathrm{x}}^{\Delta}[k]). \tag{6}$$

Next, we convert the masked STFTs to the time-domain responses, $h_{\mathrm{m}}[n]$ and $h_{\mathrm{s}}[n]$. We obtain the desired FIR $h_{\mathrm{r}}[n]$ by converting the mid/side to stereo. We apply channel-wise convolutions to the input $u[n]$ and get the output signal $y[n]$. The FFT and hop lengths are set to 384 and 192, respectively. The parameter vector p_{r} has the size of 768 (2 channels, each with 2 filters, and 192 magnitude values for each filter). We briefly discuss some alternatives to this processor. First, we could use other artificial reverberation models [16]. But again, we chose this STFT-based FIR due to its simplicity and fast computation. Or, we could use a single FIR to represent both reverb and multitap delay since they are linear time-invariant [25]. However, we decided to separate those two for interpretable, controllable, and more compact representations.

B.5. Compressor

We implement the canonical feed-forward digital compressor [26]. First, for a given input audio, we sum the left and right channels to obtain a mid signal $u_{\mathrm{m}}[n]$. Then, we calculate its energy envelope $G_u[n] = \log g_u[n]$ where

$$g_u[n] = \alpha[n] g_u[n-1] + (1 - \alpha[n]) u_{\mathrm{m}}^2[n]. \tag{7}$$

Here, the coefficient $\alpha[n]$ is typically set to a different constant for an "attack" (increasing) and "release" (decreasing) phase:

$$\alpha[n] = \begin{cases} \alpha^{\mathrm{att}} & g_u[n] > g_u[n-1], \\ \alpha^{\mathrm{rel}} & g_u[n] \leq g_u[n-1]. \end{cases} \tag{8}$$

As this part (also known as ballistics) bottlenecks the computation speed in GPU, following the recent work [12], we restrict the constants to the same value: $\alpha = \alpha^{\mathrm{att}} = \alpha^{\mathrm{rel}}$. By doing so, Equation 7 simplifies to a one-pole IIR filter, whose length-N FIR approximation can be computed in parallel using the frequency-sampling method [15] and applying IFFT to the sampled response:

$$h^{\mathrm{env}}[n] \approx \frac{1}{N} \sum_{k=0}^{N-1} \frac{1-\alpha}{1 - \alpha w_N^{-k}} w_N^{kn}. \tag{9}$$

Therefore, the energy envelope can be simply computed as

$$g_u[n] \approx h^{\mathrm{env}}[n] * u^2[n]. \tag{10}$$

Next, we compute the compressed energy envelope $G_y[n]$. We use a quadratic knee, interpolating the compression and the bypass region. For a given threshold T and half of the knee width W,

$$G_y[n] = \begin{cases} G_y^{\mathrm{above}}[n] & G_u[n] \geq T + W, \\ G_y^{\mathrm{mid}}[n] & T - W \leq G_u[n] < T + W, \\ G_y^{\mathrm{below}}[n] & G_u[n] < T - W \end{cases} \tag{11}$$

where, for a given compression ratio R, each term is

$$G_y^{\mathrm{above}}[n] = T + \frac{G_u[n] - T}{R}, \tag{12a}$$
$$G_y^{\mathrm{mid}}[n] = G_u[n] + \left(\frac{1}{R} - 1\right) \frac{(G_u[n] - T + W)^2}{4W}, \tag{12b}$$

and $G_y^{\mathrm{below}}[n] = G_u[n]$. Finally, we can compute the output as

$$y_{\mathrm{x}}[n] = \exp(G_y[n] - G_u[n]) \cdot u_{\mathrm{x}}[n] \quad (\mathrm{x} \in \{\mathrm{l}, \mathrm{r}\}). \tag{13}$$

The scalar parameters introduced above, α, T, W, and R, are concatenated and used as a parameter vector $p_{\mathrm{c}} \in \mathbb{R}^4$. Our implementation is almost identical to the recently introduced differentiable compressor [12]. However, we smoothened to the energy $u_{\mathrm{m}}^2[n]$ instead of the gain reduction $G_y[n] - G_u[n]$ as the former showed a slightly better matching performance in the initial experiments. Since the approximated ballistics (Equation 9) could be one cause of the reduced modeling capability of loudness dynamics, it might be desirable to try alternatives that allow the different attack and release coefficients [18, 27].

B.6. Noisegate

Its implementation is the same as the compressor above, except for the gain computation part. We set $G_y^{\mathrm{above}}[n] = G_u[n]$ and

$$G_y^{\mathrm{mid}}[n] = G_u[n] + (1 - R) \frac{(G_u[n] - T - W)^2}{4W}, \tag{14a}$$
$$G_y^{\mathrm{below}}[n] = T + R(G_u[n] - T). \tag{14b}$$

B.7. Multitap Delay

We consider a 2 seconds of delay effect with at most one delay d_m at every 100ms (hence the number of delay taps is $M = 20$). Each delay is filtered with a 39-tap zero-phase FIR $c_m[n]$ parameterized in the same way as the equalizer. We use independent delays and their filters for the left and right channels, but we ignore this in the following equations for simplicity. Under this setting, the multitap delay's single-channel FIR $h_{\mathrm{d}}[n]$ is given as follows,

$$h_{\mathrm{d}}[n] = \sum_{m=1}^{M} c_m[n] * \delta[n - d_m] \tag{15}$$

where $\delta[n]$ is an unit impulse. Here, we aim to optimize each delay length $d_m \in \mathbb{N}^+$, which is discrete, using gradient descent. To this end, we exploit the fact that each delay $\delta[n - d_m]$ corresponds to a complex sinusoid in the frequency domain. Recent work showed that the sinusoid's angular frequency z_m can be optimized with the gradient descent when we allow it to be inside of the unit disk [28]. In short, for each complex parameter $z_m \in \mathbb{C}$ where $|z_m| \leq 1$, we compute a surrogate damped sinusoid and apply IFFT to it.

$$\delta[n - d_m] \approx \frac{1}{N} \sum_{k=0}^{N-1} z_m^k w_N^{kn}. \tag{16}$$

This relaxed FIR is not exactly the discrete delay signal; precisely, it is a lowpassed version of an aliased sinc kernel. Hence, we use it only for backpropagation with the straight-through technique [29]. Also, we normalize each gradient and regularize the parameter z_m to be closer to the unit circle. We empirically observed that this improves the performance. To summarize, each complex conjugate gradient is modified as follows and used for the optimization:

$$\frac{\partial L}{\partial z_m^*} \leftarrow \mathrm{sgn}\left(\sum_n \frac{\partial L}{\partial h_{\mathrm{d}}[n]} \frac{\partial \tilde{h}_{\mathrm{d}}[n]}{\partial z_m^*} \right) + \gamma(|z_m| - 1)\,\mathrm{sgn}(z_m^*) \tag{17}$$

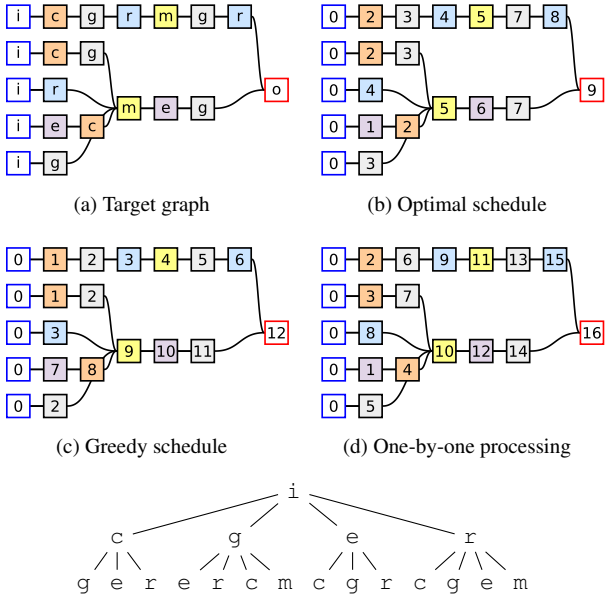
where $\tilde{h}_{\mathrm{d}}[n]$ is the surrogate FIR obtained with the continuous delay, $\mathrm{sgn}(z) = z/|z|$, and γ is a regularization strength set to 0.01. This multitap delay has parameter p_{d} of size 880 (20 delays each for the left and right channel; each delay with a real and imaginary part of the sinusoid's angular frequency and 20 log-magnitude bins of the FIR filter). Note that we restricted the delay lengths to be in separate time intervals; we failed to optimize this processor without this constraint. Resolving this is left as a future work.

C. AUDIO PROCESSING GRAPHS IN GPU

We slightly deviate from the main text and consider a more general setup. We do not distinguish between the processors and auxiliary modules; both are referred to as nodes. We omit all the weights $\mathbf{w}$ for simplicity. Then, each node's output y_i is given as follows,

$$y_i = f_i(u_i, p_i), \quad u_i = \sum_{j \in \mathcal{N}^+(i)} y_j \tag{18}$$

where $\mathcal{N}^+(i)$ denotes a set of source node indices for the node v_i. In short, each output y_i is obtained by gathering the input signals, aggregating those, and processing the sum with the parameters (the latter two can be omitted according to the node type). Note that a batch of multiple graphs can be seen as a single large disconnected graph. Therefore, it is enough to consider only a single graph case. The remaining structural restrictions to graphs are (i) being acyclic and (ii) having only single-input single-output (SISO) systems.



(a) Target graph (b) Optimal schedule

(c) Greedy schedule (d) One-by-one processing

(e) All possible sequences visualized up to depth 2

Figure 1: Various batched processing schedules. For each schedule 1b-1d, the processing orders are shown inside the nodes.

C.1. Batched Processing

We may obtain the final graph output $\mathbf{Y}$ by simply processing each node one by one. Nevertheless, we can accelerate the computation via parallelism. Consider a subset sequence $V_0, \cdots, V_N \subset V$ that satisfies the followings,

(i) It is a *partition*: $\cup_n V_n = V$ and $V_n \cap V_m = \emptyset$ for $n \neq m$.

(ii) The sequence must be *causal*: there is no path from $u \in V_n$ to $v \in V_m$ if $n \geq m$.

(iii) Each subset V_n is *homogeneous*: it has only one type t_n.

Then, we can compute a batch of output signals $\mathbf{Y}_n$ of each subset V_n sequentially, from $n = 0$ to N. Consequently, we reduce the number of the gather-aggregate-process iterations from $|V|$ to N. Figure 1 shows an example. For a graph with $|V| = 21$ nodes (1a), we can find a sequence with $N = 9$ (1b). Intuitively, this batched processing is effective for a graph with fewer types and a structure that resembles the (pruned version of) mixing console, e.g., having the identical processor order for each track.

C.2. Node Type Scheduling

To maximize the parallelism, we want to find the shortest sequence. This is a variant of the scheduling problem [30]. First, we always choose a maximal subset V_i when the type t_i is fixed. This makes the subset sequence equivalent to a type string, e.g., `iecgrmegro` for 1b. We also choose the initial and the last subset, V_0 and V_N, to collect all input and output nodes, respectively. Since the search tree for the shortest sequence exponentially grows, the brute-force search is too expensive for most graphs (1e). Fortunately, we know the optimal schedule for the graphs in this paper as they follow the fixed ordering. If we do not have such restrictions or prior assumptions about the graph structure, we may try the greedy method that chooses a type with the largest number of computable nodes (1c). However, this usually results in a longer sequence.

Algorithm 1 Batch computation of audio processing graphs.

Input: Node types $\mathbf{T}$, edges $\mathbf{E}$, parameters $\mathbf{P}$, and inputs $\mathbf{S}$
Output: Output signals $\mathbf{Y}$ and (optional) intermediate signals $\mathbf{U}$

1: $\bar{\mathbf{T}}, N \leftarrow \mathrm{ScheduleBatchedProcessing}(\mathbf{T}, \mathbf{E})$
2: $\sigma \leftarrow \mathrm{OptimizeNodeOrder}(\bar{\mathbf{T}}, \mathbf{T}, \mathbf{E})$
3: $\mathbf{T}, \mathbf{E}, \mathbf{P} \leftarrow \mathrm{Reorder}(\sigma, \mathbf{T}, \mathbf{E}, \mathbf{P})$
4: $\mathbf{I}^{\mathrm{G}}, \mathbf{I}^{\mathrm{P}}, \mathbf{I}^{\mathrm{A}}, \mathbf{I}^{\mathrm{S}} \leftarrow \mathrm{GetReadWriteIndex}(\bar{\mathbf{T}}, \mathbf{T}, \mathbf{E}, \mathbf{P})$
5: $\mathbf{U} \leftarrow \mathrm{Initialize}(\mathbf{S}, \mathbf{T})$
6: **for** $n \leftarrow 1$ to N **do**
7: $\quad \bar{\mathbf{U}}_n \leftarrow \mathrm{Gather}(\mathbf{U}, \mathbf{I}^{\mathrm{G}}_n)$ ▷ `index_select`
8: $\quad \mathbf{U}_n \leftarrow \mathrm{Aggregate}(\bar{\mathbf{U}}_n, \mathbf{I}^{\mathrm{A}}_n)$ ▷ `scatter`
9: $\quad \mathbf{P}_n \leftarrow \mathrm{Gather}(\mathbf{P}[\bar{t}_n], \mathbf{I}^{\mathrm{P}}_n)$ ▷ `slice`
10: $\quad \mathbf{Y}_n \leftarrow \mathrm{Process}(\bar{t}_n, \mathbf{U}_n, \mathbf{P}_n)$
11: $\quad \mathbf{U} \leftarrow \mathrm{Store}(\mathbf{U}, \mathbf{Y}_n, \mathbf{I}^{\mathrm{S}}_n)$ ▷ `slice`
12: **end for**
13: $\mathbf{Y} \leftarrow \mathbf{Y}_N$
14: **return** $\mathbf{Y}, \mathbf{U}$

C.3. Implementation Details

Inputs — We represent a graph with a node type vector $\mathbf{T} \in \mathbb{N}^{|V|}$ and an edge index tensor $\mathbf{E} \in \mathbb{N}^{2\times|E|}$. Its parameters are collected in a dictionary $\mathbf{P}$ whose key is a type t and value is its corresponding parameter tensor given as $\mathbf{P}[t] \in \mathbb{R}^{|V_t|\times N_t}$ where $|V_t|$ and N_t are the number of nodes and parameters. All sources are stacked to a single tensor $\mathbf{S} \in \mathbb{R}^{K\times 2\times L}$. We ensure that all the inputs, $\mathbf{T}$, $\mathbf{E}$, $\mathbf{P}$, and $\mathbf{S}$, share a node order. For example, a k^{th} source s_k must correspond to the first k^{th} input `i` in the type list $\mathbf{T}$. Likewise, an l^{th} type-t parameter $\mathbf{P}[t]_l \in \mathbb{R}^{N_t}$ must correspond to the first l^{th} type t in the type list $\mathbf{T}$.

Preprocessing — Algorithm 1 obtains the output $\mathbf{Y} \in \mathbb{R}^{|V_\circ|\times 2\times L}$ from the prescribed inputs. First, we schedule the batched processings, resulting in a type list $\bar{\mathbf{T}} \in \mathbb{N}^N$ (1). Next, as the main batched processing loop (6-12) contains multiple memory read/writes, we calculate the node reordering σ that achieves contiguous memory accesses and improves the computation speed (2). This procedure allows memory accesses via `slice`, as shown in the comments of Algorithm 1. After permuting the graph tensors (3), we retrieve lists of tensor indices, $\mathbf{I}^{\mathrm{G}}$, $\mathbf{I}^{\mathrm{P}}$, $\mathbf{I}^{\mathrm{A}}$, and $\mathbf{I}^{\mathrm{S}}$, used for the read/writes in the main loop (4). Note that all these preprocessing steps (1-4) are done in CPU. Then, we create an intermediate output tensor $\mathbf{U} \in \mathbb{R}^{|V|\times 2\times L}$ (5). As we put all the inputs to be the first partition V_0, it can be initialized with simple concatenation: $\mathbf{U} = \mathbf{S} \oplus \mathbf{0}$.

Main loop — The remainings repeat batched processing and necessary read/writes (6-12). For each n^{th} iteration, we collect the previous outputs $\bar{\mathbf{U}}_n$ that are routed to the current partition nodes. We achieve this by accessing the intermediate tensor $\mathbf{U}$ with the index $\mathbf{I}^{\mathrm{G}}_n$ with `index_select` (7). Then, we aggregate them using `scatter` operation if multiple edges are connected to some nodes (8). We can similarly obtain a parameter tensor $\mathbf{P}_n$ with its corresponding index $\mathbf{I}^{\mathrm{P}}_n$. Especially, our node reordering (3) makes this a simple `slice`, faster than the usual `index_select`. With the obtained input signals $\mathbf{U}_n \in \mathbb{R}^{|V_n|\times 2\times L}$ and parameters $\mathbf{P}_n \in \mathbb{R}^{|V_n|\times N_t}$, we batch-compute the node outputs $\mathbf{Y}_n$ (10). Then, we save them to the intermediate output tensor $\mathbf{U}$ with the `slice` index $\mathbf{I}^{\mathrm{S}}_n$ (11) so that the remaining steps can access them as inputs. After the iteration, we have all node outputs saved in $\mathbf{U}$. The final graph outputs are given as $\mathbf{Y}_N \in \mathbb{R}^{|V_N|\times 2\times L}$ since we set the last node partition V_N to collect all output nodes.

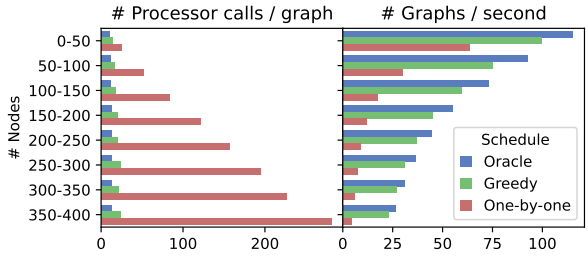


Figure 2: Benchmark results of different processing schedules.

C.4. Benchmark

We measured the efficiency of the schedules with the pruned graphs on a single `RTX3090` GPU. Figure 2 reports the result.

- Our pruned graphs require at most 14 batched processings when scheduled correctly. The optimal schedule achieves 11.8 calls on average, while the greedy and one-by-one schedule report 17.5 and 77.6 calls, respectively. The latter especially increases the processor calls linearly to the graph size.
- As a result, the batched processing improves the speed across all graph sizes (especially for large ones). On average, the one-by-one computes 27.8 graph outputs per second. The greedy and optimal schedule achieve 67.2 and 81.2, respectively, indicating that improved scheduling methods for arbitrary graphs could be desirable for further research.
- Note that the speed-up from the batched processing is sublinear to the graph size. For graphs with 350 to 400 nodes, the optimal calls the processors 21.1 times less than the one-by-one, but the speedup is about 5.8 fold.

C.5. Extension

We can relax the SISO restriction and allow multiple-input multiple-output (MIMO). A processor f_i with M inputs and N outputs processes the incoming signals as follows,

$$[y_{i1}, \cdots, y_{iN}]^T = f_i\big([u_{i1}, \cdots, u_{iM}]^T, p_i\big), \tag{19a}$$

$$u_{il} = \sum_{(j,k)\in\mathcal{N}^+(i,l)} y_{jk}. \tag{19b}$$

Here, $(j,k) \in \mathcal{N}^+(i,l)$ denotes a connection from node j channel k to node i channel l. In a graph viewpoint, this forms a multigraph that allows multiple edges between two nodes. The only remaining restriction (to the graphs) is being acyclic. Implemenetation-wise, we need to introduce an edge channel type tensor, replace the intermediate $\mathbf{U}$ with a node-channel flattened tensor, and reshape the processor input/outputs, $\mathbf{U}_n$ and $\mathbf{Y}_n$, appropriately to gather from and store to the intermediate $\mathbf{U}$.

D. DRY/WET PRUNING ALGORITHM

Algorithm 2 describes the details of the dry/wet method.

- For a simpler description, we modified the initialization part to include per-type node sets and weights, as in line 6-10.
- The termination condition is given in line 11.
- The trial candidate sampling is implemented in line 12-13.
- The candidate pool update is expanded to separately handle the trial successes and failures, shown in line 18 and 20-25.

		MedleyDB				MixingSecrets				Internal			
		L_{a}	L_{lr}	L_{m}	L_{s}	L_{a}	L_{lr}	L_{m}	L_{s}	L_{a}	L_{lr}	L_{m}	L_{s}
Base graph		50.7	1.45	1.42	198	7.30	2.16	2.02	22.9	1.12	.951	.940	1.63
+ Gain/panning	`g`	.550	.583	.485	.550	.876	.856	.819	.973	.642	.619	.597	.734
+ Stereo imager	`sg`	.541	.564	.483	.553	.847	.834	.791	.928	.538	.616	.595	.727
+ Equalizer	`e sg`	.450	.453	.390	.504	.700	.698	.622	.780	.522	.497	.467	.626
+ Reverb	`e sg r`	.368	.361	.360	.390	.614	.601	.579	.674	.463	.451	.432	.517
+ Compressor	`ec sg r`	.315	.304	.297	.356	.558	.542	.512	.637	.396	.377	.347	.482
+ Noisegate	`ecnsg r`	.302	.288	.281	.353	.548	.532	.502	.625	.393	.374	.343	.480
+ Multitap delay (full)	`ecnsgdr`	.296	.288	.284	.324	.545	.529	.502	.618	.385	.369	.338	.465

Table 2: Per-dataset results of the mixing consoles with different processor type configurations.

		MedleyDB									MixingSecrets									Internal								
	τ	L_{a}	ρ	ρ_{g}	ρ_{s}	ρ_{e}	ρ_{r}	ρ_{c}	ρ_{n}	ρ_{d}	L_{a}	ρ	ρ_{g}	ρ_{s}	ρ_{e}	ρ_{r}	ρ_{c}	ρ_{n}	ρ_{d}	L_{a}	ρ	ρ_{g}	ρ_{s}	ρ_{e}	ρ_{r}	ρ_{c}	ρ_{n}	ρ_{d}
`MC`	–	.296	–	–	–	–	–	–	–	–	.545	–	–	–	–	–	–	–	–	.385	–	–	–	–	–	–	–	–
`BF`	.01	.305	.63	.35	.81	.54	.77	.67	.71	.53	.566	.65	.50	.82	.43	.71	.61	.77	.75	.402	.80	.76	.92	.61	.81	.85	.87	.80
`DW`	.01	.302	.56	.32	.79	.42	.74	.57	.56	.43	.561	.57	.47	.82	.22	.62	.54	.74	.55	.397	.74	.74	.82	.49	.70	.87	.86	.61
`H`	.001	.295	.44	.22	.73	.26	.61	.50	.54	.24	.550	.43	.35	.76	.13	.42	.43	.55	.38	.388	.59	.48	.77	.40	.57	.78	.77	.39
	.01	.302	.61	.32	.81	.48	.74	.69	.71	.52	.563	.62	.49	.85	.34	.64	.60	.79	.65	.400	.77	.73	.93	.56	.75	.85	.86	.74
	.1	.375	.83	.59	.93	.83	.92	.80	.84	.90	.648	.84	.70	.91	.75	.86	.83	.93	.91	.474	.93	.90	.99	.84	.92	.93	.95	.96

Table 3: Per-dataset results of the pruning. `MC`: mixing console. `BF`: brute-force, `DW`: dry/wet, and `H`: hybrid.

Algorithm 2 Music mixing graph search (dry/wet method).

Input: A mixing console G_{c}, dry tracks $\mathbf{S}$, and mixture y
Output: Pruned graph G_{p}, parameters $\mathbf{P}$, and weights $\mathbf{w}$

$\mathbf{P}, \mathbf{w} \leftarrow \text{Initialize}(G_{\text{c}})$
$\mathbf{P}, \mathbf{w} \leftarrow \text{Train}(G_{\text{c}}, \mathbf{P}, \mathbf{w}, \mathbf{S}, y)$
$L_{\text{a}}^{\min} \leftarrow \text{Evaluate}(G_{\text{c}}, \mathbf{P}, \mathbf{w}, \mathbf{S}, y)$
$G_{\text{p}} \leftarrow G_{\text{c}}$
for $n \leftarrow 1$ **to** N_{iter} **do**
  $T_{\text{cand}} \leftarrow \text{GetProcessorTypeSet}(V)$
  for t **in** T_{cand} **do**
    $V_t, \mathbf{w}_t \leftarrow \text{Filter}(V, t), \text{Filter}(\mathbf{w}, t)$
    $N_t, r_t, \mathbf{m}_t \leftarrow |V_t|, 0.1, \mathbf{1}$
  end for
  while $T_{\text{cand}} \neq \emptyset$ **do**
    $t \leftarrow \text{SampleType}(T_{\text{cand}})$
    $\bar{V}_t, \bar{\mathbf{m}} \leftarrow \text{GetLeastWeightNodes}(V_t, \mathbf{w}_t, \lfloor N_t r_t \rceil)$
    $L_{\text{a}} \leftarrow \text{Evaluate}(G_{\text{p}}, \mathbf{P}, \mathbf{w} \odot \mathbf{m} \odot \bar{\mathbf{m}}, \mathbf{S}, y)$
    if $L_{\text{a}} < L_{\text{a}}^{\min} + \tau$ **then**
      $L_{\text{a}}^{\min} \leftarrow \min(L_{\text{a}}^{\min}, L_{\text{a}})$
      $\mathbf{m} \leftarrow \mathbf{m} \odot \bar{\mathbf{m}}$
      $V_t \leftarrow V_t \setminus \bar{V}_t$
    else
      if $N_t > 1$ **then**
        $r_t \leftarrow r_t / 2$
      else
        $T_{\text{cand}} \leftarrow T_{\text{cand}} \setminus \{t\}$
      end if
    end if
  end while
  $G_{\text{p}}, \mathbf{P}, \mathbf{w} \leftarrow \text{Prune}(G_{\text{p}}, \mathbf{P}, \mathbf{w}, \mathbf{m})$
  $\mathbf{P}, \mathbf{w} \leftarrow \text{Train}(G_{\text{p}}, \mathbf{P}, \mathbf{w}, \mathbf{S}, y)$
end for
return $G_{\text{p}}, \mathbf{P}, \mathbf{w}$

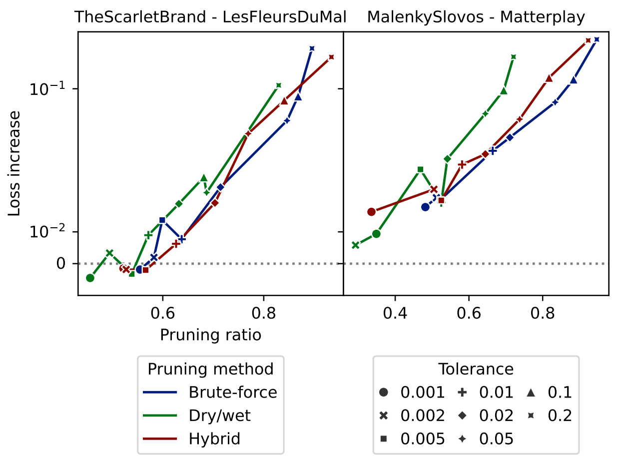


Figure 3: Loss increases from the mixing console and remaining processor ratios for different pruning methods and tolerances.

E. SUPPLEMENTARY RESULTS

- Table 2 and 3 report the per-dataset results on the mixing consoles and graph pruning, respectively.
- Figure 3 compares the pruning methods on 2 random-sampled songs using 7 tolerance settings from 0.001 to 0.2.
- Figure 4 shows multiple graphs obtained by pruning the same console (song) repeatedly.
- Refer to Figure 5-7 for more pruned graphs obtained with the default setting — hybrid method and $\tau = 0.01$.
- Figure 9-11 show more spectrogram plots.

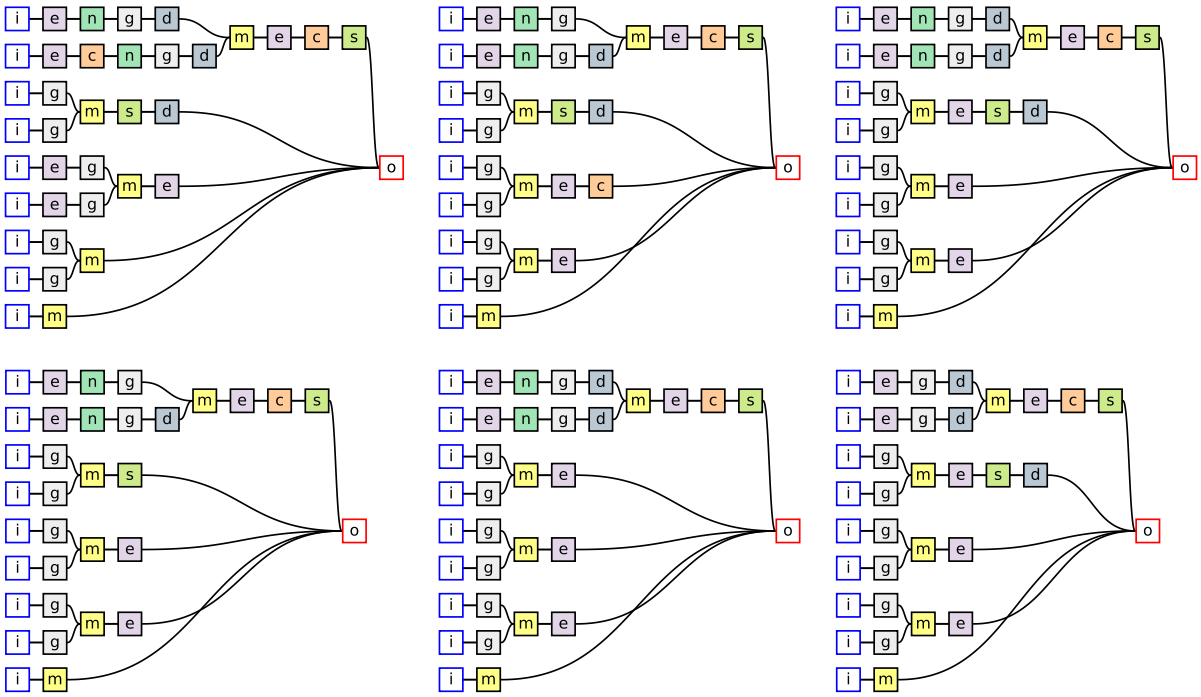
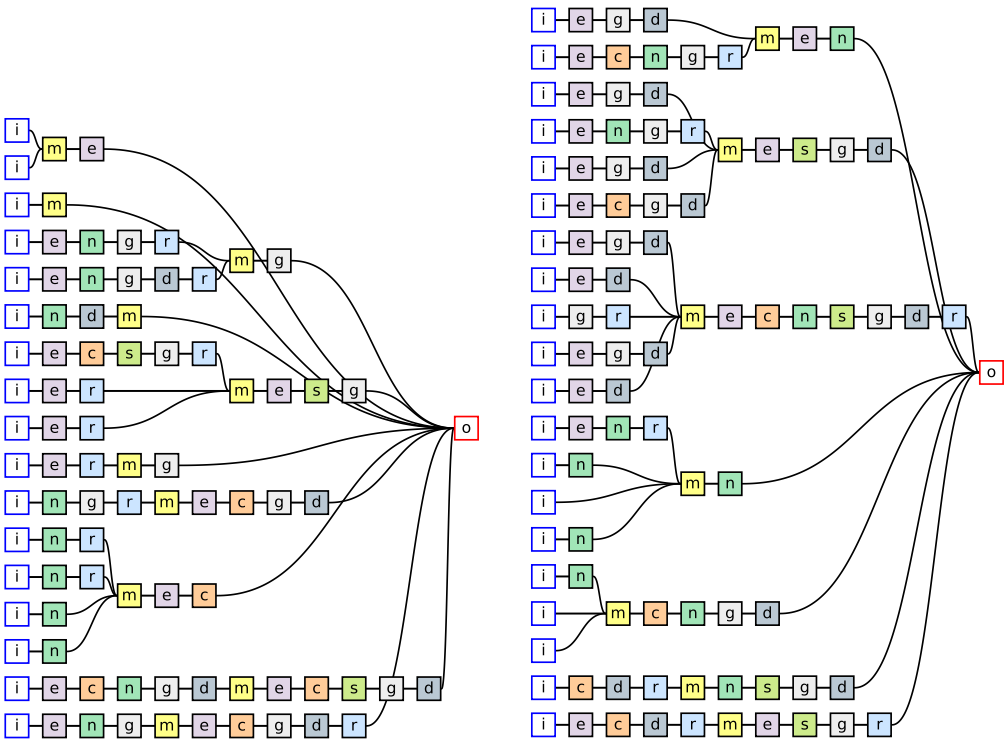
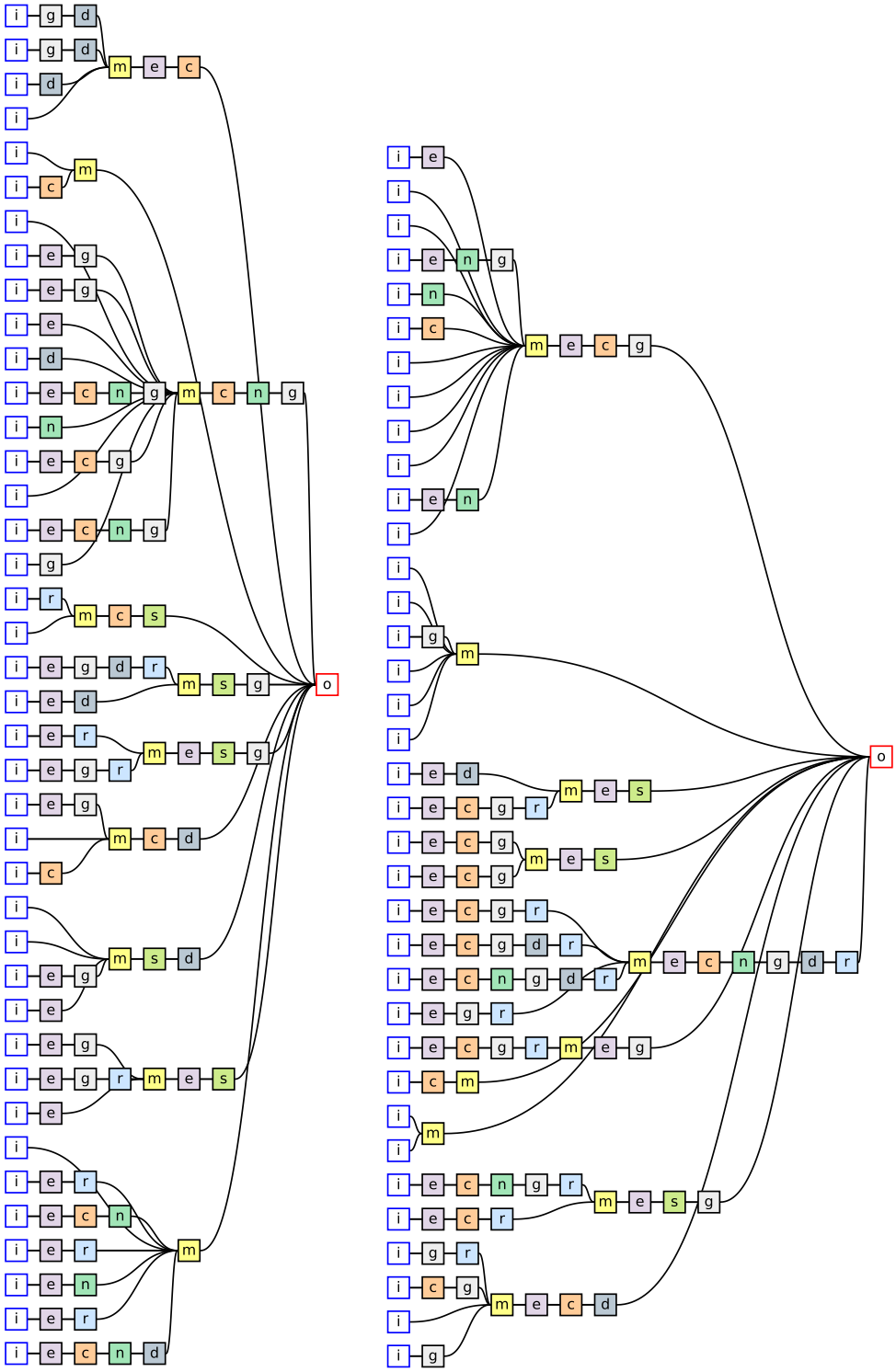


Figure 4: Each pruning run (default setting) yields a slightly different graph. Song: `EthanHein_GirlOnABridge`.



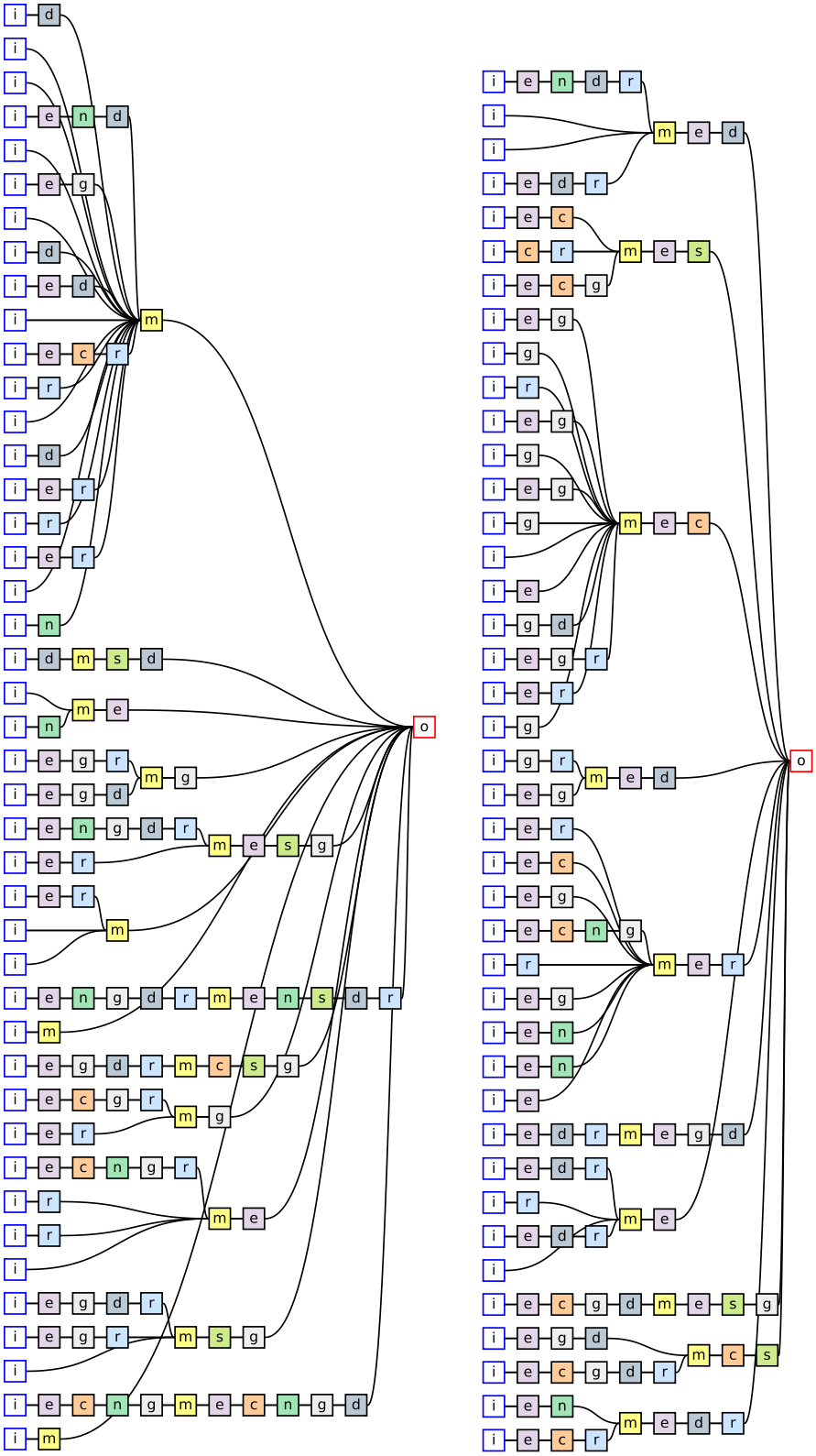
(a) `Internal_part1_65536` (b) `DonCamilloChoir_TrudeTheBumblebee`

Figure 5: Example pruned graphs (default setting). the number of tracks: $K \leq 20$.



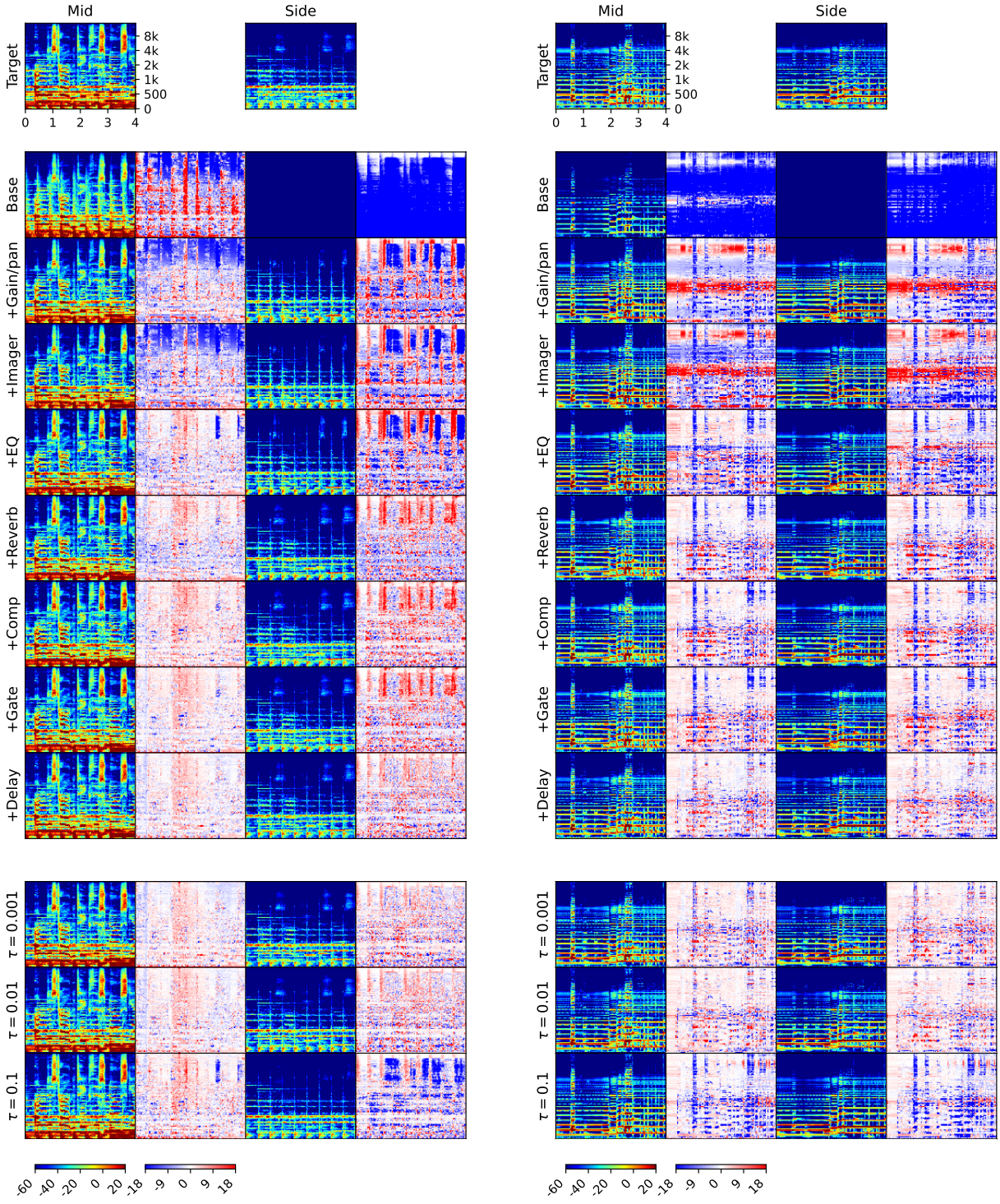
(a) `LittleTybee_TheAlchemist` (b) `RaftMonk_Tiring`

Figure 6: Example pruned graphs (default setting). the number of tracks: $K > 20$.



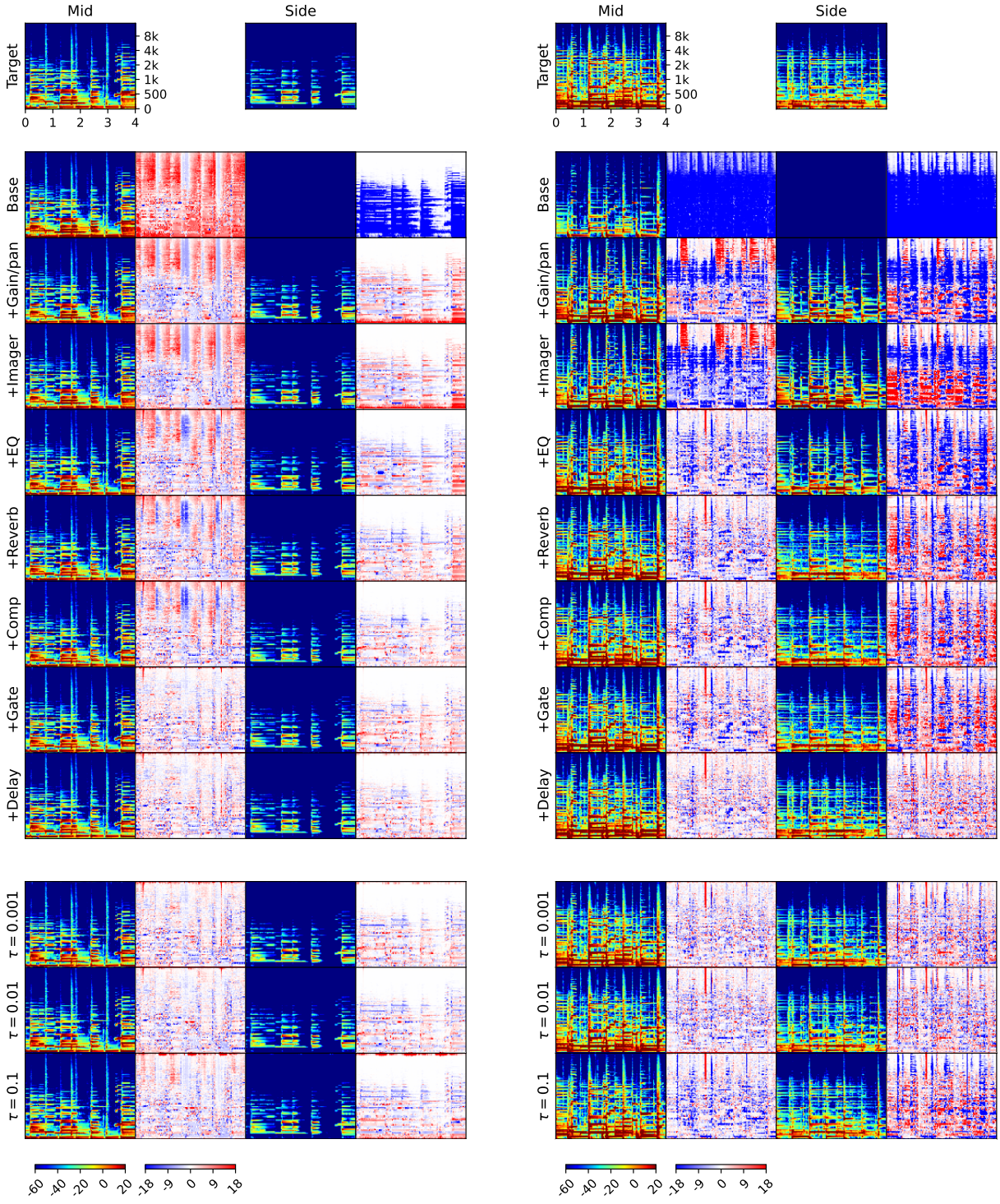
(a) `Internal_part2_67692` (b) `StevenClark_Bounty`

Figure 7: Example pruned graphs (default setting). the number of tracks: $K > 20$ (continued).



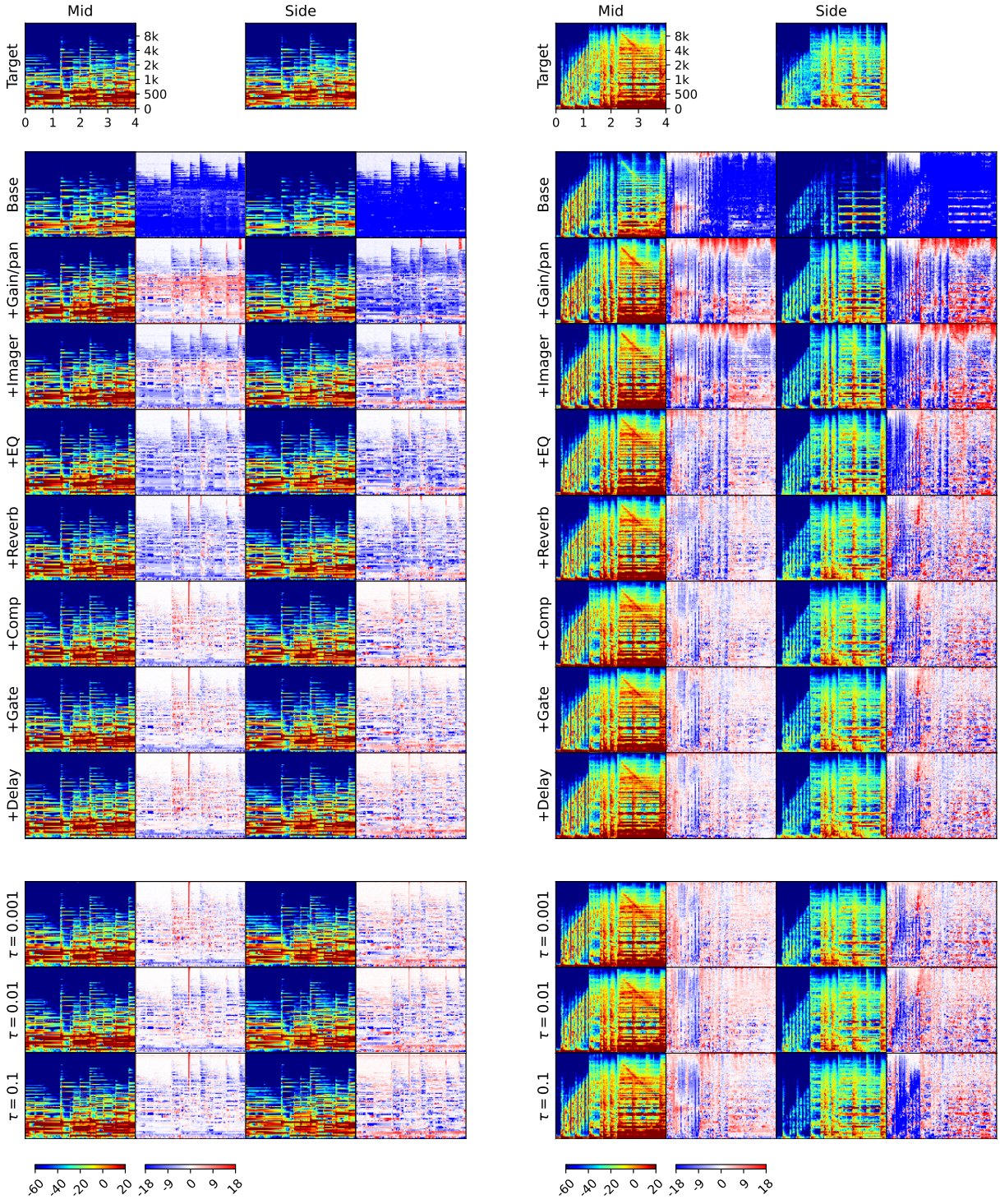
(a) `Torres_NewSkin` (b) `TablaBreakbeatScience_RockSteady`

Figure 8: Matching of target music mixes with mixing consoles and their pruned versions: `MedleyDB` dataset.



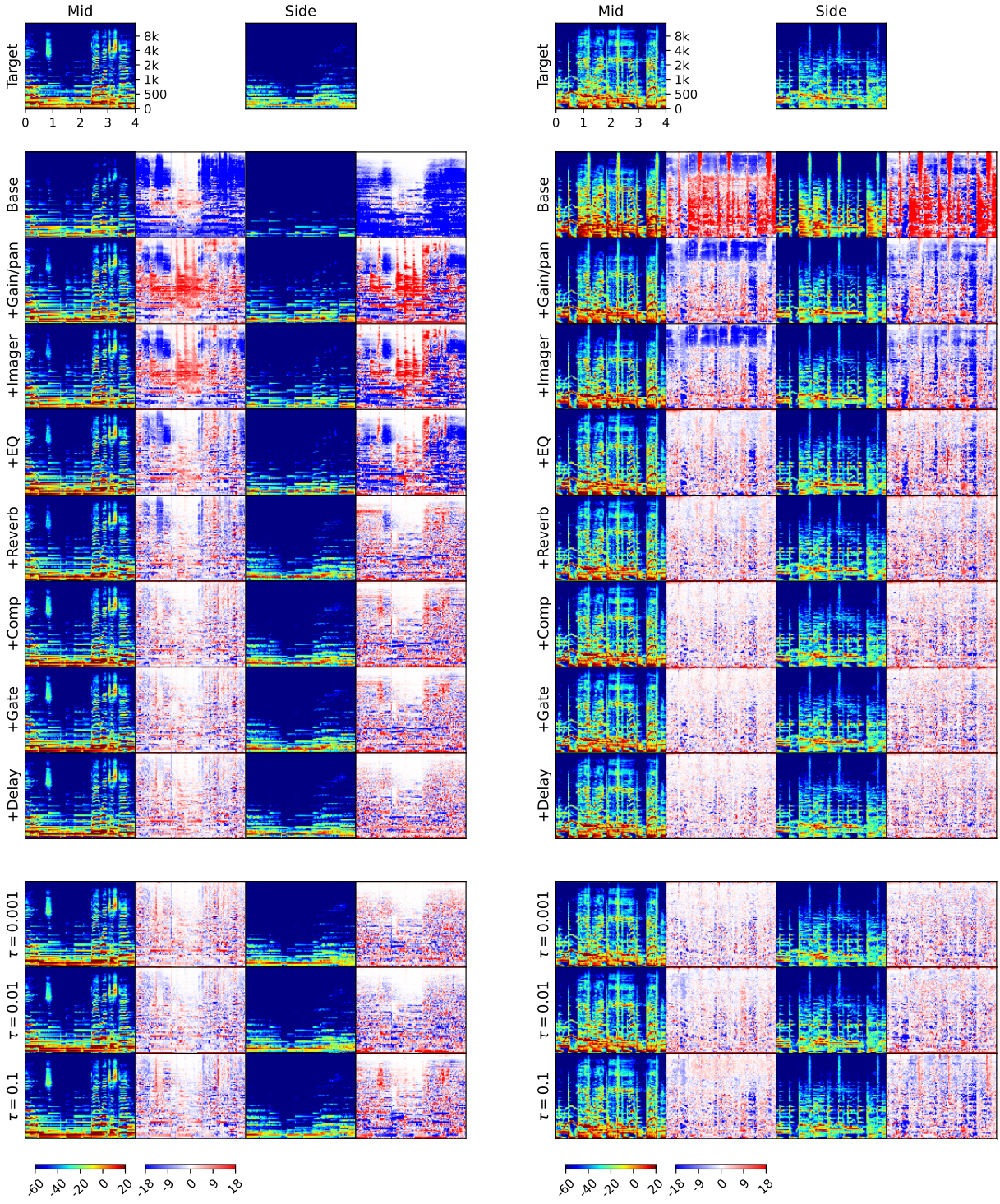
(a) `MusicDelta_SwingJazz` (b) `ChrisJacoby_BoothShotLincoln`

Figure 9: Matching of target music mixes with mixing consoles and their pruned versions: `MedleyDB` dataset.



(a) `HowlProject_IfIWereABell` (b) `IanDearden_TeraniaCreekWalking`

Figure 10: Matching of target music mixes with mixing consoles and their pruned versions: `MixingSecrets` dataset.



(a) `Internal_66680` (b) `Internal_67954`

Figure 11: Matching of target music mixes with mixing consoles and their pruned versions: `Internal` dataset.

F. REFERENCES

[1] S. Lee, J. Park, S. Paik, and K. Lee, "Blind estimation of audio processing graph," in *IEEE ICASSP*, 2023.

[2] C. Mitcheltree and H. Koike, "SerumRNN: Step by step audio VST effect programming," in *Artificial Intelligence in Music, Sound, Art and Design*, 2021.

[3] J. Guo and B. McFee, "Automatic recognition of cascaded guitar effects," in *DAFx*, 2023.

[4] J. Colonel, "Music production behaviour modelling," 2023.

[5] C. J. Steinmetz, J. Pons, S. Pascual, and J. Serrà, "Automatic multitrack mixing with a differentiable mixing console of neural audio effects," in *IEEE ICASSP*, 2021.

[6] N. Uzrad *et al.*, "DiffMoog: a differentiable modular synthesizer for sound matching," *arXiv:2401.12570*, 2024.

[7] J. Engel, L. H. Hantrakul, C. Gu, and A. Roberts, "DDSP: differentiable digital signal processing," in *ICLR*, 2020.

[8] M. A. Martínez-Ramírez, O. Wang, P. Smaragdis, and N. J. Bryan, "Differentiable signal processing with black-box audio effects," in *IEEE ICASSP*, 2021.

[9] Z. Ye, W. Xue, X. Tan, Q. Liu, and Y. Guo, "NAS-FM: Neural architecture search for tunable and interpretable sound synthesis based on frequency modulation," *arXiv:2305.12868*, 2023.

[10] F. Caspe, A. McPherson, and M. Sandler, "DDX7: Differentiable FM synthesis of musical instrument sounds," in *ISMIR*, 2022.

[11] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *NeurIPS*, 2019.

[12] C. J. Steinmetz, N. J. Bryan, and J. D. Reiss, "Style transfer of audio effects with differentiable signal processing," *JAES*, vol. 70, no. 9, 2022.

[13] J. T. Colonel, M. Comunità, and J. Reiss, "Reverse engineering memoryless distortion effects with differentiable waveshapers," in *AES Convention 153*, 2022.

[14] A. Carson, S. King, C. V. Botinhao, and S. Bilbao, "Differentiable grey-box modelling of phaser effects using frame-based spectral processing," in *DAFx*, 2023.

[15] S. Nercessian, "Neural parametric equalizer matching using differentiable biquads," in *DAFx*, 2020.

[16] S. Lee, H.-S. Choi, and K. Lee, "Differentiable artificial reverberation," *IEEE/ACM TASLP*, vol. 30, 2022.

[17] B. Hayes, J. Shier, G. Fazekas, A. McPherson, and C. Saitis, "A review of differentiable digital signal processing for music & speech synthesis," *Frontiers in Signal Process.*, 2023.

[18] C. J. Steinmetz, T. Walther, and J. D. Reiss, "High-fidelity noise reduction with differentiable signal processing," in *AES Convention 155*, 2023.

[19] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *Journal of Machine Learning Research*, vol. 20, no. 55, 2019.

[20] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *ICLR*, 2019.

[21] D. C. Elton, Z. Boukouvalas, M. D. Fuge, and P. W. Chung, "Deep learning for molecular design—a review of the state of the art," *Molecular Systems Design & Engineering*, vol. 4, no. 4, 2019.

[22] J. You, B. Liu, Z. Ying, V. Pande, and J. Leskovec, "Graph convolutional policy network for goal-directed molecular graph generation," *NeurIPS*, 2018.

[23] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *arXiv preprint arXiv:1611.01578*, 2016.

[24] X. Serra and J. Smith, "Spectral modeling synthesis: A sound analysis/synthesis based on a deterministic plus stochastic decomposition," *Computer Music Journal*, vol. 14, 1990.

[25] J. T. Colonel and J. Reiss, "Reverse engineering a nonlinear mix of a multitrack recording," *Journal of the AES*, vol. 71, no. 9, 2023.

[26] D. Giannoulis, M. Massberg, and J. D. Reiss, "Digital dynamic range compressor design—a tutorial and analysis," *JAES*, vol. 60, no. 6, 2012.

[27] J. Colonel, J. D. Reiss, *et al.*, "Approximating ballistics in a differentiable dynamic range compressor," in *AES Convention 153*, 2022.

[28] B. Hayes, C. Saitis, and G. Fazekas, "Sinusoidal frequency estimation by gradient descent," in *IEEE ICASSP*, 2023.

[29] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv:1308.3432*, 2013.

[30] J. D. Ullman, "NP-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, 1975.