GRAFX: AN OPEN-SOURCE LIBRARY FOR AUDIO PROCESSING GRAPHS IN PYTORCH

Sungho Lee[†*], *Marco A. Martínez-Ramírez*[♮], *Wei-Hsiang Liao*[♮], *Stefan Uhlich*[♯], *Giorgio Fabbro*[♯], *Kyogu Lee*[†], *and Yuki Mitsufuji*[♮♭]

[†]Department of Intelligence and Information, Seoul National University, Seoul, South Korea
[♮]Sony AI, Tokyo, Japan [♯]Sony Europe B.V., Stuttgart, Germany [♭]Sony Group Corporation, Tokyo, Japan

ABSTRACT

We present `GRAFX`, an open-source library designed for handling audio processing graphs in `PyTorch`. Along with various library functionalities, we describe technical details on the efficient parallel computation of input graphs, signals, and processor parameters in GPU. Then, we show its example use under a music mixing scenario, where parameters of every differentiable processor in a large graph are optimized via gradient descent. The code is available at `https://github.com/sh-lee97/grafx`.

1. INTRODUCTION

Motivation — Under the umbrella of so-called *differentiable signal processing* [1, 2], numerous attempts have been made to import existing audio processors to automatic differentiation frameworks, e.g., `PyTorch` [3]. Differentiable processors, as standalone modules, allow gradient-based optimization of their parameters. Alternatively, they can be used to train a neural network as a parameter estimator [4, 5]. In either case, as the processors are identical to or approximate the real-world ones, the obtained parameters are easy to interpret and control. To further leverage this advantage of differentiable signal processing, it would be desirable to consider the *composition* of processors since it is a standard real-world practice. In a more general setting, this composition can be represented in a *graph* format [6]. Yet, there are few public implementations [1, 7] that provide highly flexible graph-related functionalities.

Contributions — In response, we present a library called `GRAFX` that allows users to handle audio processing graphs and their applications in `PyTorch`. Along with the open-source code, this demonstration paper serves multiple purposes. First, we highlight the library's core components and applications. This includes our custom data structures that allow the creation and modification of graphs, computation of output signals, and potential use as input of graph neural networks (GNNs) [6]. Second, we dive into the heart of `GRAFX`, an optimized processing algorithm that computes the output audio from graphs, source audio, and processor parameters. Unlike the previous implementations [1, 7], ours allows the change of the graph for every optimization step. This is useful in multiple scenarios, e.g., training a GNN that predicts the parameters of given graphs [6] or pruning a graph with gradient descent [8]. Also, we use batched node processing faster than the conventional "one-by-one" computation of processors. Third, our library is complemented with various differentiable processors; we report their technical details. Finally, we describe how we used `GRAFX` to create the music mixing graphs in our companion paper [8] and evaluate the speedup we obtain with the batched node processing.

[*] Work partially done during an internship at Sony AI.

2. AUDIO PROCESSING GRAPHS IN GPU

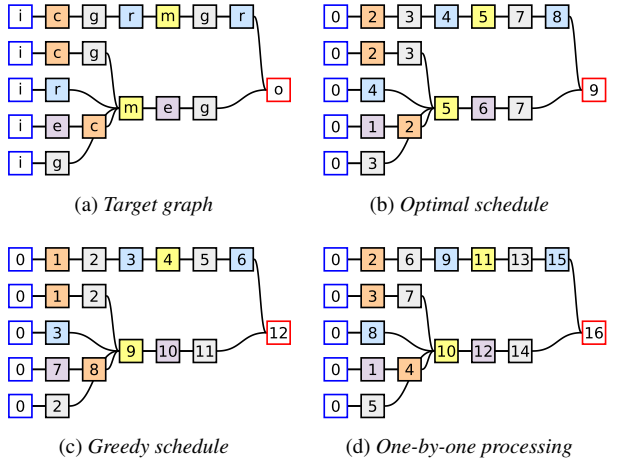
Definitions — We write an audio processing graph as $G = (V, E)$ where V and E are the node and edge set, respectively. Each node $v_i \in V$ can represent a processor f_i with a type t_i, e.g., reverb `r`. It takes M signal(s) $u_i[n]$ and a collection of parameter tensors p_i as input and produces N output(s) $y_i[n]$ (n denotes a time index).

$$y_i^{(1)}[n], \cdots, y_i^{(N)}[n] = f_i\big(u_i^{(1)}[n], \cdots, u_i^{(M)}[n], p_i\big). \quad \text{(1a)}$$

$$u_i^{(l)}[n] = \sum_{(j,k)\in\mathcal{N}^+(i,l)} y_j^{(k)}[n]. \quad \text{(1b)}$$

Here, $\mathcal{N}^+(i,l)$ denotes a collection of nodes and channel indices that send their output signals to l^{th} input of node i. With this setup, each edge $e_{ij} \in E$ becomes a "cable" connecting two nodes. Note that each edge also requires a type attribute $t_{ij} = (k, l)$, a tuple of input and output channel indices, unless every processor in a graph is a single-input single-output system (SISO), i.e., $M = N = 1$. In short, each node's outputs can be computed by finding its inputs, aggregating those, and processing the sums with the parameters. The graph output can be obtained by repeating this procedure over all nodes in topological order, starting from input nodes `i` until we reach the output nodes `o`. We allow any graph except those with cycles, as feedback loops are hard to resolve and bottleneck the processing speed due to the forced sample-level recursion.

Graph representations — There are two main use scenarios when handling the audio processing graphs. First, users may create and modify a graph. For this case, we provide a mutable data structure `GRAFX` (same as the library name). It inherits `MultiDiGraph` class from `networkx` [9] and provides additional functionalities, e.g., adding a serial chain of processors. Second, each graph can be used to compute output audio or fed into a GNN. In such cases, representing each graph as a collection of tensors is more convenient and efficient. Therefore, we provide a `GRAFXTensor` class, which is compatible with `Data` class from `torch_geometric` [10]. The following are the tensors we use to describe each graph. First, we have a node type vector $\mathbf{T}_V \in \mathbb{N}^{|V|}$, an edge index tensor $\mathbf{E} \in \mathbb{N}^{2\times|E|}$, and an (optional) edge type tensor $\mathbf{T}_E \in \mathbb{N}^{2\times|E|}$ where $|\cdot|$ denotes the size of a given set. All parameters are collected in a dictionary $\mathbf{P}$ whose key is a node type t and value $\mathbf{P}[t]$ contains the parameters of that type. In this paper, we assume that the value is a single tensor in a form $\mathbf{P}[t] \in \mathbb{R}^{|V_t|\times N_t}$ where $|V_t|$ and N_t are the number of nodes and parameters. In fact, we allow $\mathbf{P}[t]$ to be a tensor with more dimensions or even a dictionary of tensors; we omit such cases for simplicity. All sources are stacked to a single tensor $\mathbf{S} \in \mathbb{R}^{K\times C\times L}$ where K, C, and L are the number of sources, channels, and length, respectively. We ensure that all the tensors, $\mathbf{T}_V$, $\mathbf{T}_E$, $\mathbf{E}$, $\mathbf{P}$, and $\mathbf{S}$, share the same node order. For example, a k^{th} source $s_k[n]$ must correspond to the first k^{th} input `i` in the node type list $\mathbf{T}_V$. Likewise, an l^{th} type-t parameter $\mathbf{P}[t]_l \in \mathbb{R}^{N_t}$ must correspond to the l^{th} type t in the type list $\mathbf{T}_V$.



(a) *Target graph* (b) *Optimal schedule*

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

`i`: *input,* `o`: *output,* `m`: *mix,* `e`: *equalizer,* `c`: *compressor,* `g`: *gain/panning.*

Figure 1: *Various schedules for batched node processing. For each schedule, the processing orders are shown inside the nodes.*

Batched processing — For faster computation in GPU, maximizing the parallelism is desirable, and batched processing is the most standard approach. Note that we have 3 levels of batched processing. First, we may want *source-level* parallelism, i.e., processing batches of multiple input sources with a single graph. This can be easily achieved when every processor supports batched processing. Next, we can consider *node-level* parallelism, processing multiple nodes with the same type simultaneously. Specifically, consider a sequence of $N+1$ node subsets $V_0, \cdots, V_N \subset V$ satisfying the following conditions.

(i) It forms a *partition*: $\cup_n V_n = V$ and $V_n \cap V_m = \emptyset$ if $n \neq m$.
(ii) It is *causal*: no path from $u \in V_n$ to $v \in V_m$ exists if $n \geq m$.
(iii) Each subset V_n is *homogeneous*: it has only a single 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 (we have no processings for $n = 0$ as V_0 contains input modules). Figure 1 shows an example. For a graph with $|V| = 21$ nodes (1a), we can obtain a sequence with $N = 9$ (1b). Finally, equipped with this batched node processing, we can also achieve *graph-level* parallelism; we can simply treat a batch of multiple graphs as a single large disconnected graph. Therefore, we will focus on the batched *node* processing for the remainder of this section.

Type scheduling — For a maximized node-level parallelism, we want to find the shortest node subset sequence. This is a variant of the scheduling problem. 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 first and the last subset, V_0 and V_N, to have all of the 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. Instead, 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, thus slower processing. We can alleviate this issue with the beam search, i.e., keeping multiple best schedules as candidates instead of one. Intuitively, the batched node processing is effective for graphs with fewer types and a certain structure, e.g., ones that apply processors in the same order for every input (e.g., see Figure 2b).

Algorithm 1 *Batch computation of audio processing graphs.*

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

1: $\bar{\mathbf{T}}, N \leftarrow \mathrm{ScheduleBatchedProcessing}(\mathbf{T}_V, \mathbf{E})$
2: $\sigma \leftarrow \mathrm{OptimizeNodeOrder}(\bar{\mathbf{T}}, \mathbf{T}_V, \mathbf{E})$
3: $\mathbf{T}_V, \mathbf{E}, \mathbf{P} \leftarrow \mathrm{Reorder}(\sigma, \mathbf{T}_V, \mathbf{E}, \mathbf{P})$
4: $\mathbf{I}^{\mathrm{G}}, \mathbf{I}^{\mathrm{P}}, \mathbf{I}^{\mathrm{A}}, \mathbf{I}^{\mathrm{S}} \leftarrow \mathrm{GetReadWriteIndex}(\bar{\mathbf{T}}, \mathbf{T}_V, \mathbf{E}, \mathbf{T}_E, \mathbf{P})$
5: $\mathbf{U} \leftarrow \mathrm{Initialize}(\mathbf{S}, \mathbf{T}_V)$
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** $\mathbf{Y}, \mathbf{U}$

Implementation details — Algorithm 1 obtains the output $\mathbf{Y}$ from the prescribed inputs (inside the following parentheses denote the line numbers). First, we schedule the batched node processing and obtain a node type list $\bar{\mathbf{T}} \in \mathbb{N}^{N+1}$ (1). Next, as the main batched processing loop (6-12) contains multiple memory reads/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 reordering the graph tensors with σ (3), we retrieve lists of indices, $\mathbf{I}^{\mathrm{G}}$, $\mathbf{I}^{\mathrm{P}}$, $\mathbf{I}^{\mathrm{A}}$, and $\mathbf{I}^{\mathrm{S}}$, used for the tensor read/writes in the main loop (4). Note that all these steps (1-4) are done in CPU and, in most cases, in multiple separate threads. Therefore, they do not bottleneck the GPU and optimization. After the preprocessing, we create an intermediate output tensor $\mathbf{U}$, which will have a shape of $|V| \times C \times L$ if all processors are SISO systems or $N_{\mathrm{sum}} \times C \times L$ where N_{sum} denotes the total number of outputs in the graph (5). As we put all the inputs to be the first partition V_0, it can be initialized with simple concatenation: $\mathbf{U} = \mathbf{S} \oplus \mathbf{0}$. The remaining repeats batched processing and necessary reads/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` 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 $\mathbf{U}_n \in \mathbb{R}^{|V_n| \times C \times L}$ 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 output of the last step $\mathbf{Y} = \mathbf{Y}_N \in \mathbb{R}^{|V_N| \times C \times L}$ since we set the last node partition V_N to collect all output nodes.

3. DIFFERENTIABLE AUDIO PROCESSORS

We report the differentiable audio processors that we provide. By default, they accept and produce stereo signals, i.e., $C = 2$. All the hyperparameters, e.g., the number of filter taps, are ones used in the companion paper [8]. We use `FlashFFTConv` [11] for every causal convolution to speed up the processing and save memory.

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.

$$y[n] = \exp(p_{\mathrm{g}}) \cdot u[n]. \tag{2}$$

Stereo imager — We multiply the side signal, i.e., left minus right, with a gain parameter $p_{\mathrm{s}} \in \mathbb{R}$ to control the stereo width. The mid and side outputs are given as

$$y_{\mathrm{m}}[n] = u_{\mathrm{l}}[n] + u_{\mathrm{r}}[n], \tag{3a}$$
$$y_{\mathrm{s}}[n] = \exp(p_{\mathrm{s}}) \cdot (u_{\mathrm{l}}[n] - u_{\mathrm{r}}[n]). \tag{3b}$$

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

Equalizer — We use a single-channel 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

$$h_{\mathrm{e}}[n] = v^{\mathrm{Hann}}[n] \cdot \frac{1}{N} \sum_{k=0}^{N-1} \exp p_{\mathrm{e}}[k] \cdot w_N^{kn} \tag{4}$$

where $-(N+1)/2 \le n \le (N+1)/2$ 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,

$$y_{\mathrm{x}}[n] = u_{\mathrm{x}}[n] * h_{\mathrm{e}}[n] \quad (\mathrm{x} \in \{\mathrm{l}, \mathrm{r}\}). \tag{5}$$

We set the FIR length to $N = 2047$. Therefore, the parameter p_{e} has a size of 1024.

Reverb — We use a variant of the filtered noise model [1]. First, we create 2 seconds of uniform noises, $u_{\mathrm{m}}[n]$ and $u_{\mathrm{s}}[n]$. Next, we apply a magnitude mask $M_{\mathrm{x}}[k, m]$ to each noise's STFT $U_{\mathrm{x}}[k, m]$ as follows,

$$H_{\mathrm{x}}[k, m] = U_{\mathrm{x}}[k, m] \odot M_{\mathrm{x}}[k, m] \quad (\mathrm{x} \in \{\mathrm{m}, \mathrm{s}\}). \tag{6}$$

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

$$M_{\mathrm{x}}[k, m] = \exp(H_{\mathrm{x}}^0[k] + (m-1)H_{\mathrm{x}}^{\Delta}[k]). \tag{7}$$

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 $y[n]$. The FFT and hop lengths are 384 and 192, respectively. The parameter p_{r} has a size of 768 (2 channels, each with 2 filters with 192 magnitudes).

Compressor — We implement the canonical feed-forward digital compressor [12]. 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

$$g_u[n] = \alpha[n] g_u[n-1] + (1 - \alpha[n]) u_{\mathrm{m}}^2[n]. \tag{8}$$

Here, the coefficient $\alpha[n]$ is typically set to a different constant for an "attack" (where $g_u[n]$ increases) and "release" (where $g_u[n]$ decreases) phase. As this part (also known as ballistics) bottlenecks the computation speed in GPU, following the recent work [5], we restrict the coefficients to the same value α. By doing so, Equation 8 simplifies to a one-pole IIR filter, whose impulse response up to a finite length N can be exactly obtained in parallel as follows,

$$h^{\mathrm{env}}[n] = (1-\alpha)\alpha^n. \tag{9}$$

Therefore, the energy envelope $g_u[n]$ can be simply computed as a convolution between the FIR $h^{\mathrm{env}}[n]$ and the energy signal $u_{\mathrm{m}}^2[n]$. Next, we calculate 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,

$$G_y[n] = \begin{cases} G_y^{\mathrm{above}}[n] & G_u[n] \ge T + W, \\ G_y^{\mathrm{mid}}[n] & T - W \le G_u[n] < T + W, \\ G_y^{\mathrm{below}}[n] & G_u[n] < T - W \end{cases} \tag{10}$$

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

$$G_y^{\mathrm{above}}[n] = T + \frac{G_u[n] - T}{R}, \tag{11a}$$
$$G_y^{\mathrm{mid}}[n] = G_u[n] + \Big(\frac{1}{R} - 1\Big) \frac{(G_u[n] - T + W)^2}{4W}, \tag{11b}$$

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

$$y_{\mathrm{x}}[n] = \exp(G_y[n] - G_u[n]) \cdot u_{\mathrm{x}}[n] \quad (\mathrm{x} \in \{\mathrm{l}, \mathrm{r}\}). \tag{12}$$

The scalar parameters introduced above, α, T, W, and R, are concatenated and used as a parameter vector $p_{\mathrm{c}} \in \mathbb{R}^4$.

Noisegate — It is identical to the compressor above, except for the gain computation: we set $G_y^{\mathrm{above}}[n] = G_u[n]$ and

$$G_y^{\mathrm{mid}}[n] = G_u[n] + (1 - R) \frac{(G_u[n] - T - W)^2}{4W}, \tag{13a}$$
$$G_y^{\mathrm{below}}[n] = T + R(G_u[n] - T). \tag{13b}$$

Multitap delay — We use a 2 seconds of delay effect with at most one delay d_m at every 100ms (therefore, the number of delay taps is $M = 20$). Each delay is filtered with an FIR $c_m[n]$, parameterized in the same way as the zero-phase equalizer but with 39 taps. Separate delays and filters are used for each left and right channel, but we will omit this for simplicity. Under this setting, the multitap delay's FIR is given as follows,

$$h_{\mathrm{d}}[n] = \sum_{m=1}^{M} c_m[n] * \delta[n - d_m] \tag{14}$$

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 complex angular frequency $z_m \in \mathbb{C}$ can be optimized with the gradient descent when we allow it to be inside the unit disk, i.e., $|z_m| \le 1$ [13]. We leverage this finding; for each delay, we compute a damped sinusoid with the angular frequency z_m. Then, we use its inverse FFT as a surrogate of the delay signal.

$$\delta[n - d_m] \approx \frac{1}{N} \sum_{k=0}^{N-1} z_m^k w_N^{kn}. \tag{15}$$

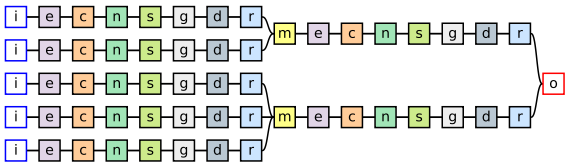
As this signal only approximates the exact delay, we use it only for backpropagation with the straight-through estimation [14]. Moreover, we normalize each gradient and regularize the parameter z_m to be closer to the unit circle. This processor has a parameter p_{d} of size 880 (40 delays, each delay with a complex angular frequency and 20 log-magnitudes of an equalizer).

```
1  import torch
2  from grafx import *

3  G = GRAFX()
4  in_id = G.add("in")
5  chain = ["eq", ..., "reverb"]
6  start_id, end_id = G.add_serial_chain(chain)
7  mix_id = G.add("mix")
8  G.connect(in_id, start_id)
9  G.connect(end_id, mix_id) ...

10 G_t = convert_to_tensor(G)
11 render_data = compute_render_data(G_t)
12 source = torch.rand(1, 5, 2, 2**17)
13 processors = {"eq": ZerophaseFIREqualizer(), ...
       "reverb": MideSideFilteredNoiseReverb()}
14 parameters = {"eq": torch.zeros(7, 1024), ...
       "reverb": torch.zeros(7, 768)}
15 output = render_grafx(source, processors,
       parameters, render_data)
```

(a) *An example code for creating and rendering a mixing console.*



(b) *An example mixing console, drawn with* `draw_grafx`*. The same notation as Fig. 1a plus* `n`*: noisegate,* `s`*: stereo imager,* `d`*: multitap delay.*

Figure 2: *Example usage of* `GRAFX` *for a music mixing scenario.*

4. MUSIC MIXING APPLICATIONS

Example usage — Figure 2a demonstrates how we used `GRAFX` to construct a graph called "music mixing console" in our companion paper [8]. Note that this is a modified and simplified version of the original code (inside the following parentheses denote the line numbers). First, we import the libraries (1-2). Then, we create an empty graph (3) and add the necessary nodes. We add a single input node `in` with `add` (4). We also add a serial chain of processors by passing a sequence of node types to `add_serial_chain` (5-6). To connect the serial chain with the input node, we pass the node indices returned by the node creation methods, `in_id` and `start_id`, to `connect` (8-9). This chain corresponds to the first upper left row of the full mixing console shown in Figure 2b. Repeating this procedure multiple times (which is omitted) will complete the graph. To compute its output audio, we first convert the graph to tensors (10). Then, we compute the batched node processing schedule and its indices with `compute_render_data` (15; corresponds to the line 1-4 of Algorithm 1). With the 5 stereo source signals of length 2^{17} (12), differentiable audio processors (13), and the dictionary of parameters (14), we finally obtain the graph output with `render_grafx` (15; the main loop of Algorithm 1). This `output` can be used to calculate a loss and perform the gradient descent to match the target mix.

Speed benchmark — Finally, we evaluated the efficiency of our batched node processing with various scheduling methods. We benchmarked with a single `RTX3090` GPU and the *pruned* graphs where negligible processors are removed from the mixing consoles [8]. Figure 3 reports the results. The optimal schedule achieves 11.8 processor calls on average. The beam (32 candidates), greedy, and one-by-one schedules report 12.6, 16.4 and 77.6 calls, respectively. The one-by-one especially increases the processor calls linearly to the graph size. Consequently, the batched node processing

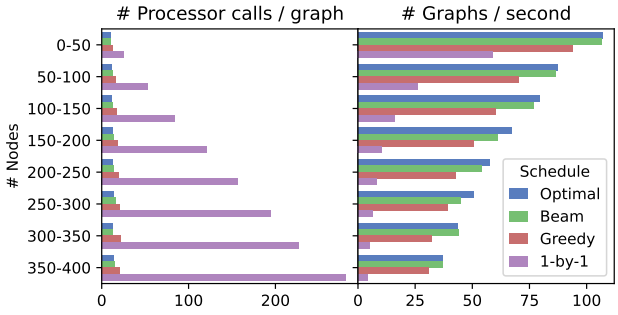


Figure 3: *Speed benchmark results of various scheduling methods.*

improves the speed across all graph sizes, especially for large ones. While the greedy method performs slightly worse than the optimal, the beam search method closes this gap.

5. CONCLUSION

Several factors will determine the usefulness of the `GRAFX` library: usability, flexibility, efficiency of the graph processing algorithms, and diversity of the provided differentiable processors. Improving these aspects and maintaining the library are left as future work.

6. REFERENCES

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

[2] 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.

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

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

[5] 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.

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

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

[8] S. Lee *et al.*, "Searching for music mixing graphs: a pruning approach," in *DAFx*, 2024.

[9] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using NetworkX," 2008.

[10] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv:1903.02428*, 2019.

[11] D. Y. Fu *et al.*, "FlashFFTConv: Efficient convolutions for long sequences with tensor cores," *ICLR*, 2023.

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

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

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