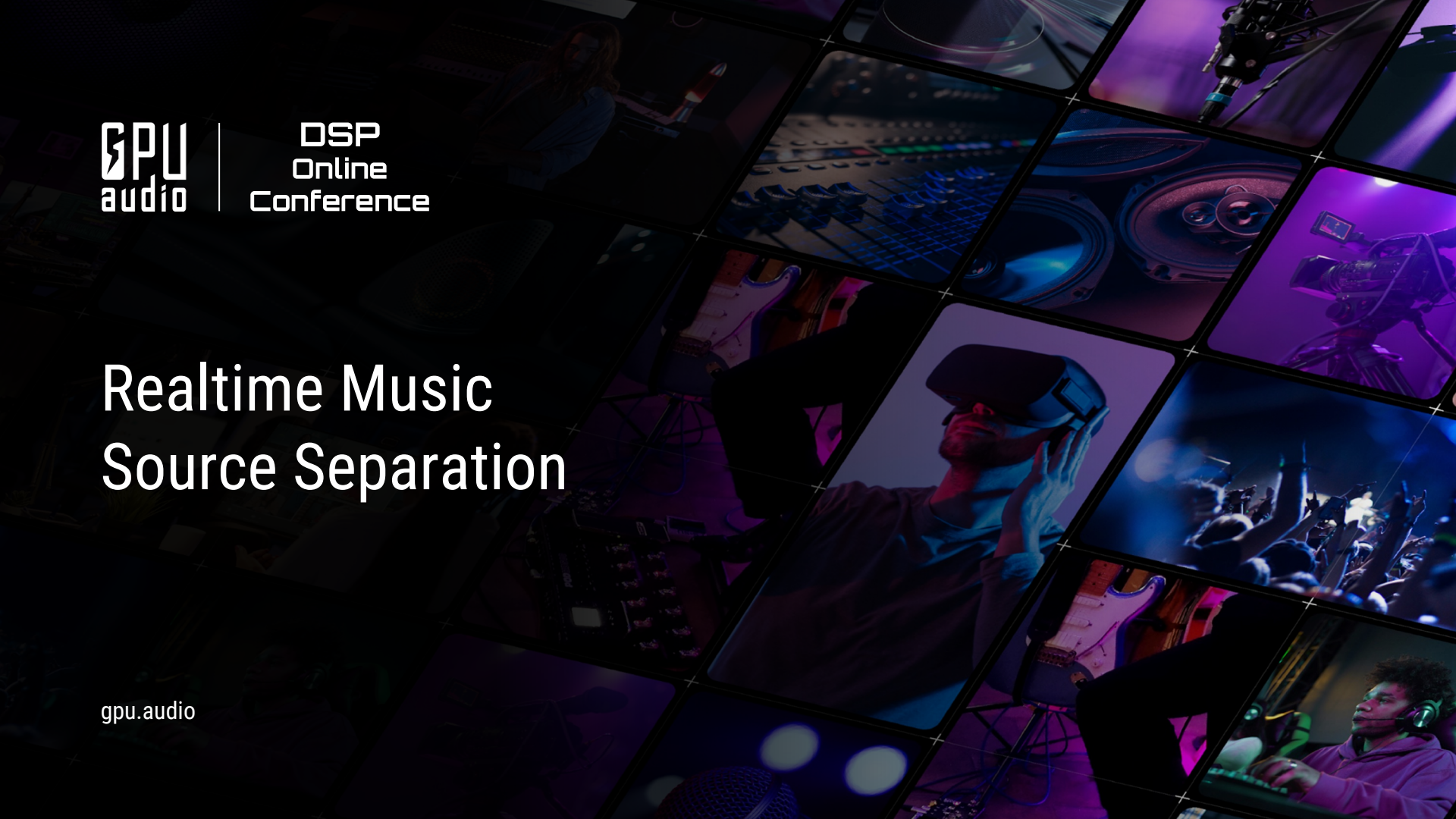
GPU audio | DSP Online Conference

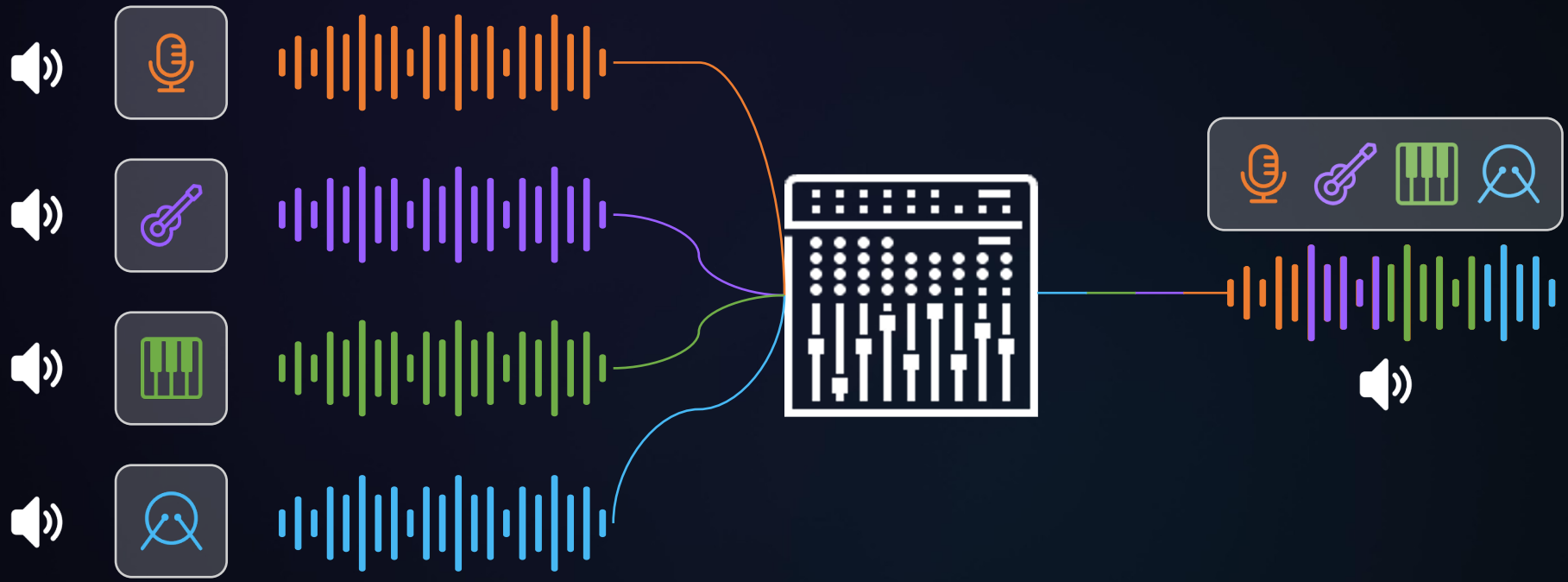
Realtime Music Source Separation



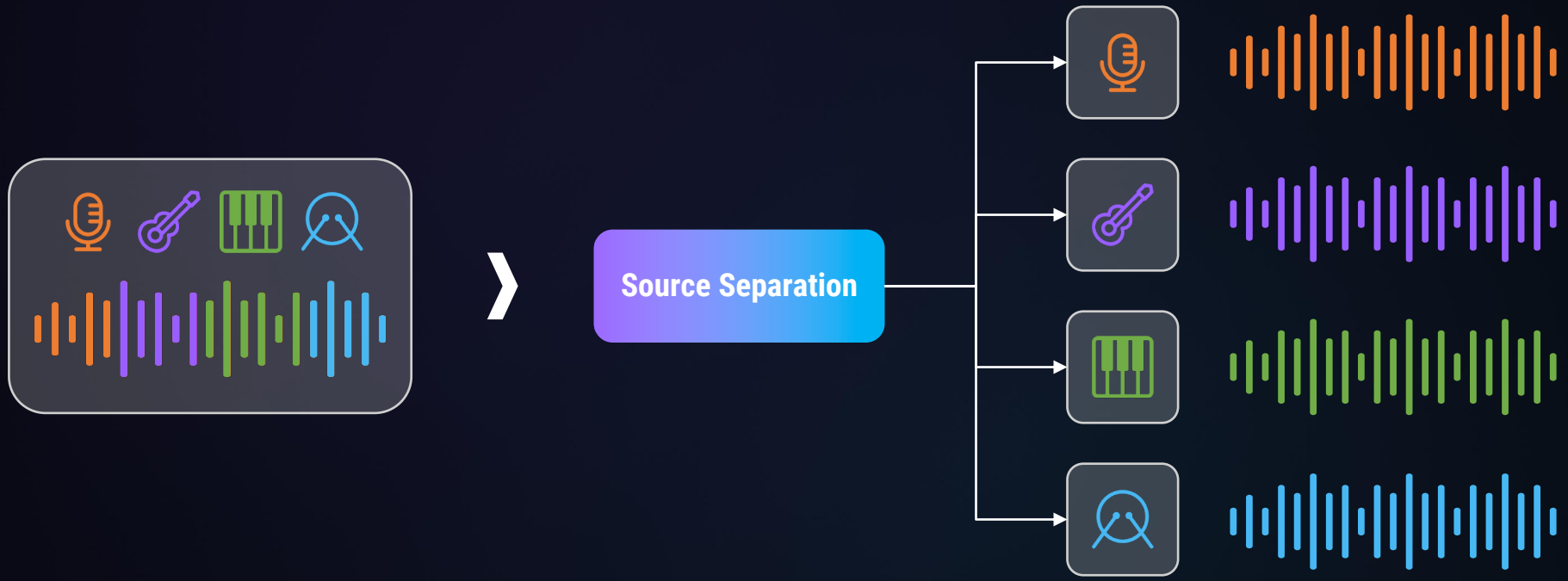
gpu.audio

Introduction

Music Production or Mixing



Music Source Separation or De-mixing



Why Sound Source Separation

Remixing and mashups

Producers can extract clean isolated stems (vocals, drums, bass) to blend them into other tracks or create entirely new arrangements

Karaoke and transcription

Separating the vocals from the instrumental accompaniment allows for the creation of karaoke versions or helps in automatically transcribing lyrics

Audio enhancement and restoration

The technology can be used to suppress unwanted sounds or remove noise, such as background hum or reverberation, making specific sounds clearer

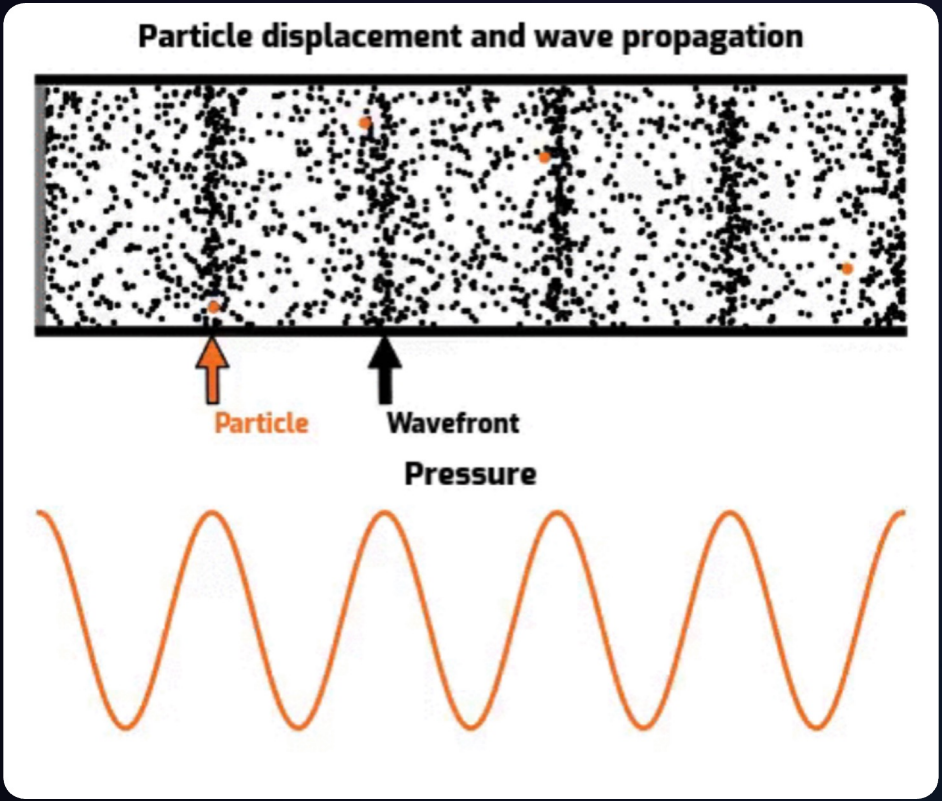
Customization and control

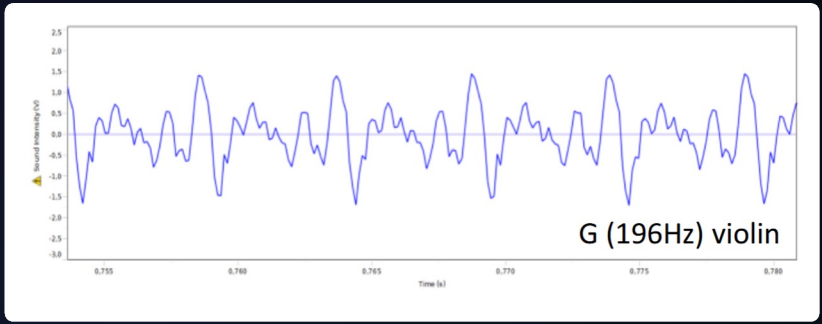
It allows for fine-grained control over the mix, such as adjusting the volume of different elements independently, which is not possible with a standard stereo track

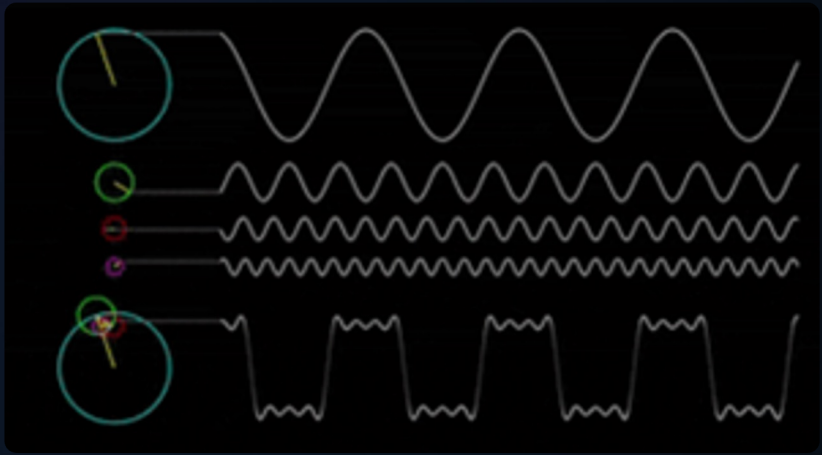
Music upmixing via sound source separation

A process that expands a stereo or mono audio recording into a multichannel spatial format by first isolating the individual instrument and vocal tracks. This technique, which leverages advanced AI and deep learning, is highly effective for converting older recordings into immersive formats like 5.1, 7.1, or Dolby Atmos

Sounds Are Just a Bunch of Sine Waves



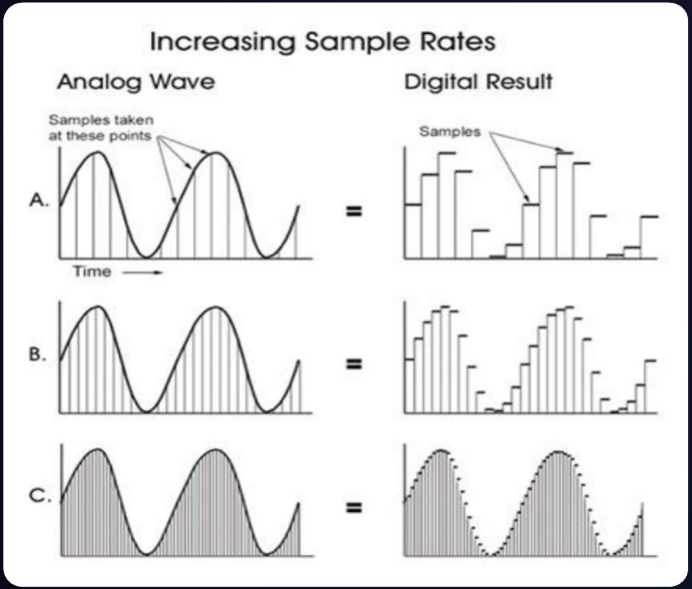


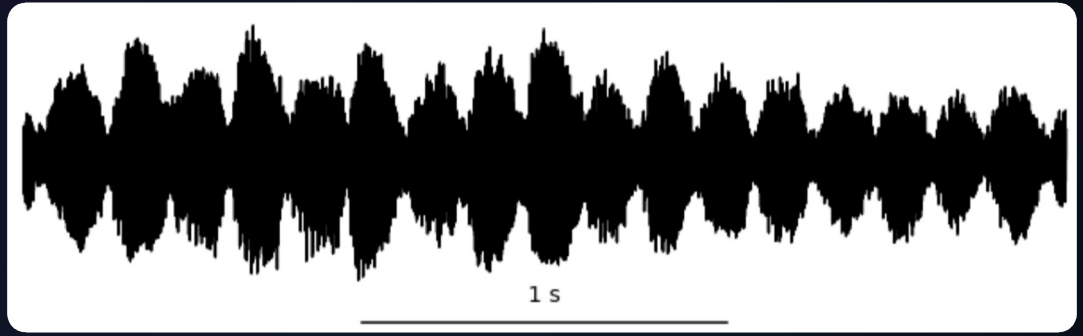


Sources: https://medium.com/age-of-awareness/the-science-behind-the-sound-10bdc94ad70

Time-Domain Representation of Audio

Digital audio is just a sequence of amplitude values sampled at a certain rate



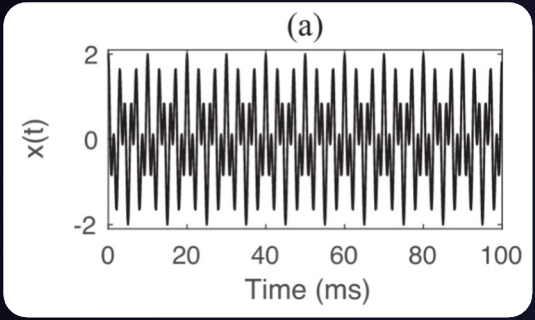


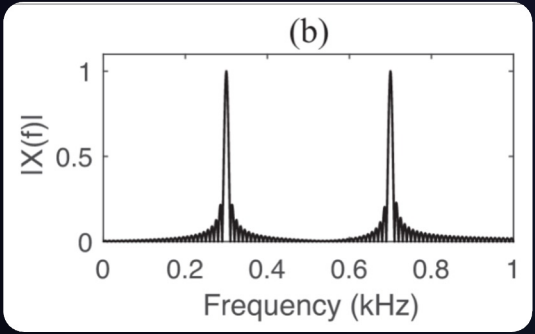
Typical Sampling Rates:

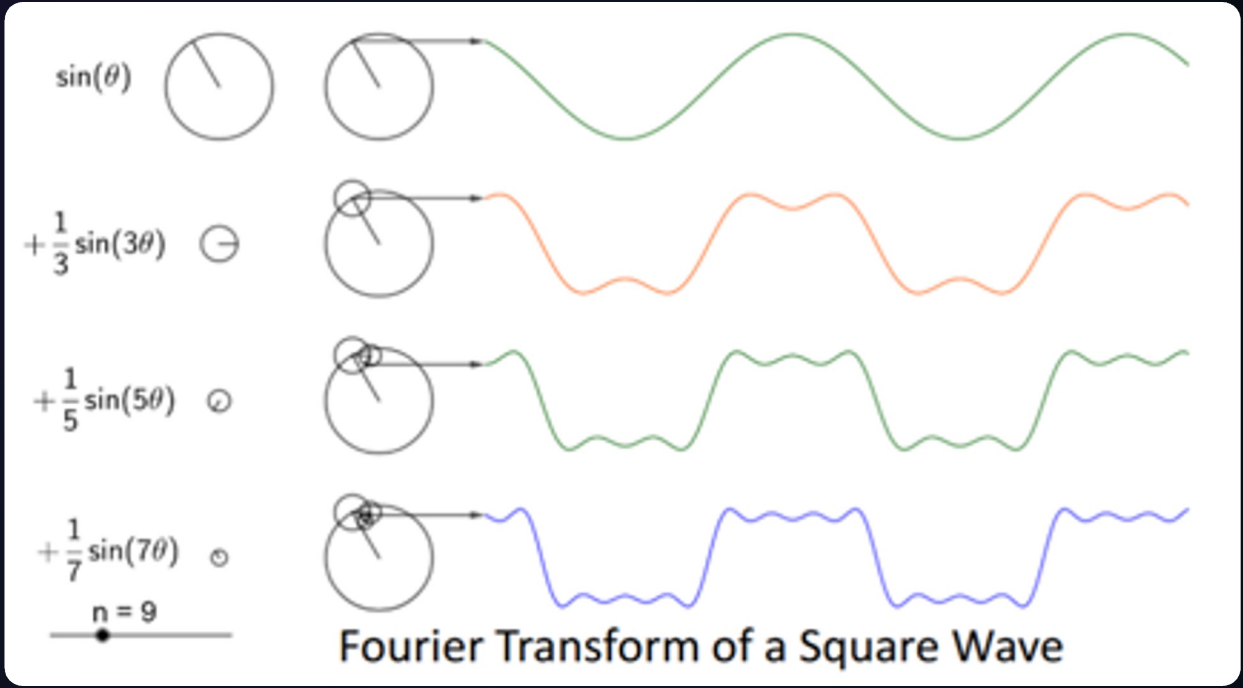
- 8kHz: walkie talkie/telephone
- 16kHz : VoIP
- 44.1kHz : CD Music
- 96kHz
- : DVD/BluRay

Sources: https://mp4gain.com/mp4gain/wp-content/uploads/2019/09/sample-rate-O.jpg
https://jvbalen.github.io/notes/waveform.html

Time-Frequency Representation of Audio

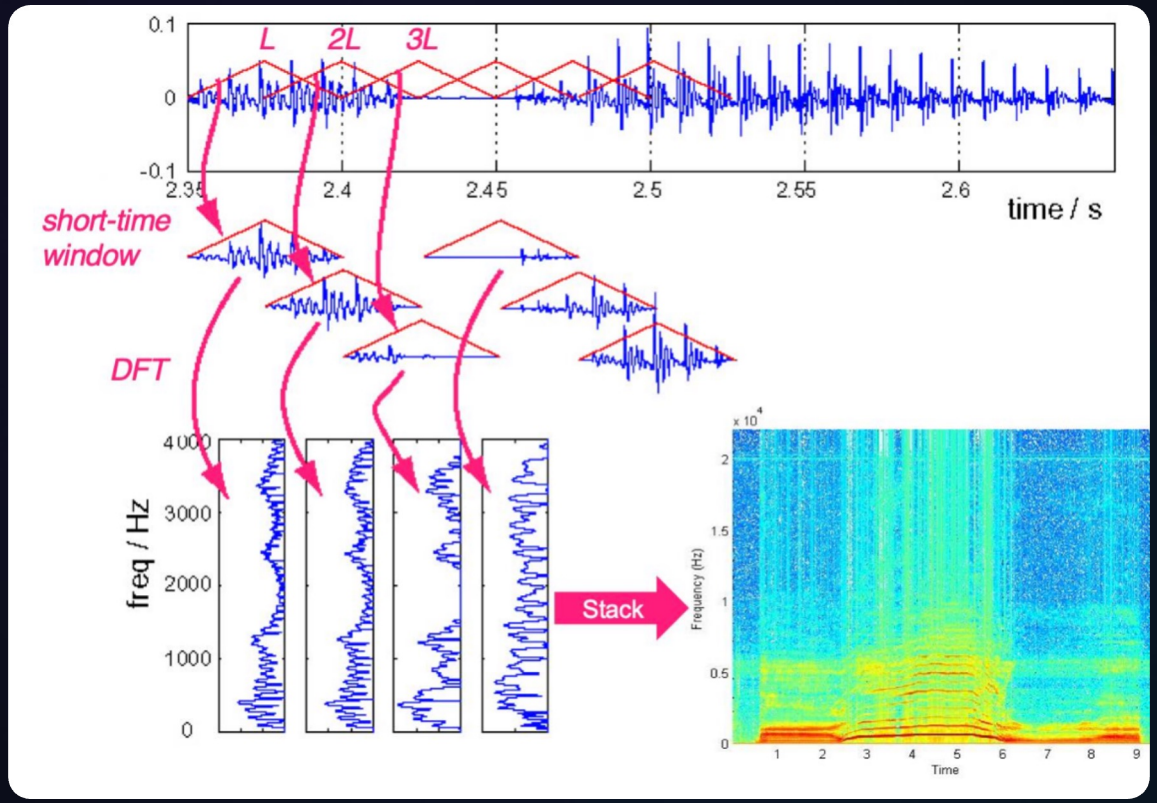






Sources: https://www.geogebra.org/resource/hwtd5jkk/dvdK7uhmI3NL7tKh/material-hwtd5jkk.png
https://source-separation.github.io/tutorial/ images/right representation.png

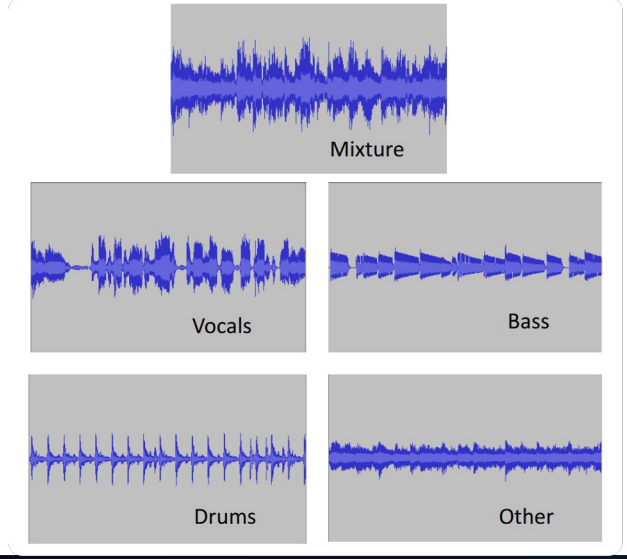
Short-time Fourier Transform



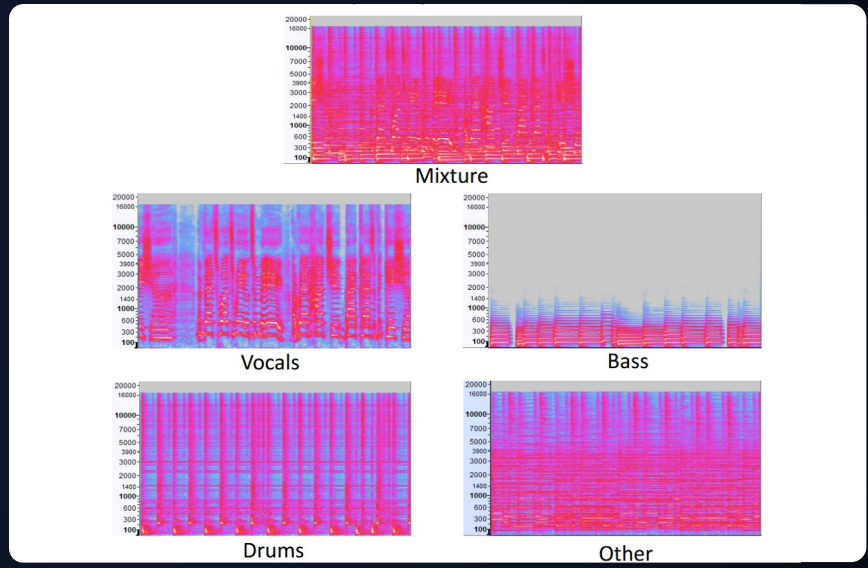
Sources: https://source-separation.github.io/tutorial/_images/stft_process.png

Two ways of Representing Sounds
Two ways of Tackling Source Separation

Time Domain or Waveform Domain



Frequency Domain



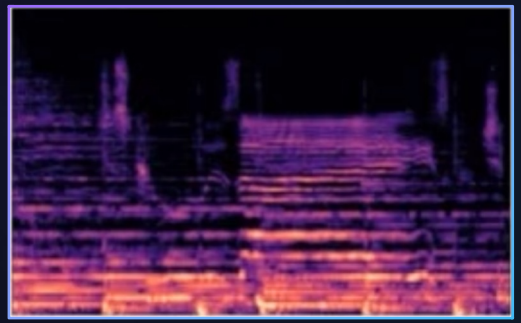
Treat Audio as a Vision Problem

Mask

0	0	1	1
1	1	0	1
0	1	0	1
0	0	1	0

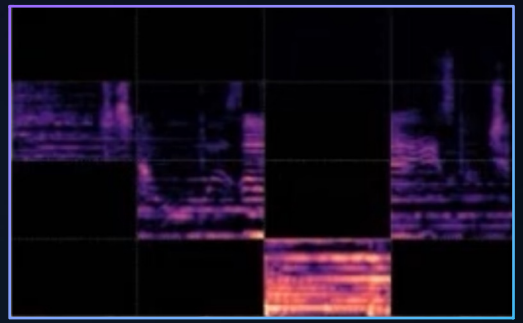
×

Spectrogram

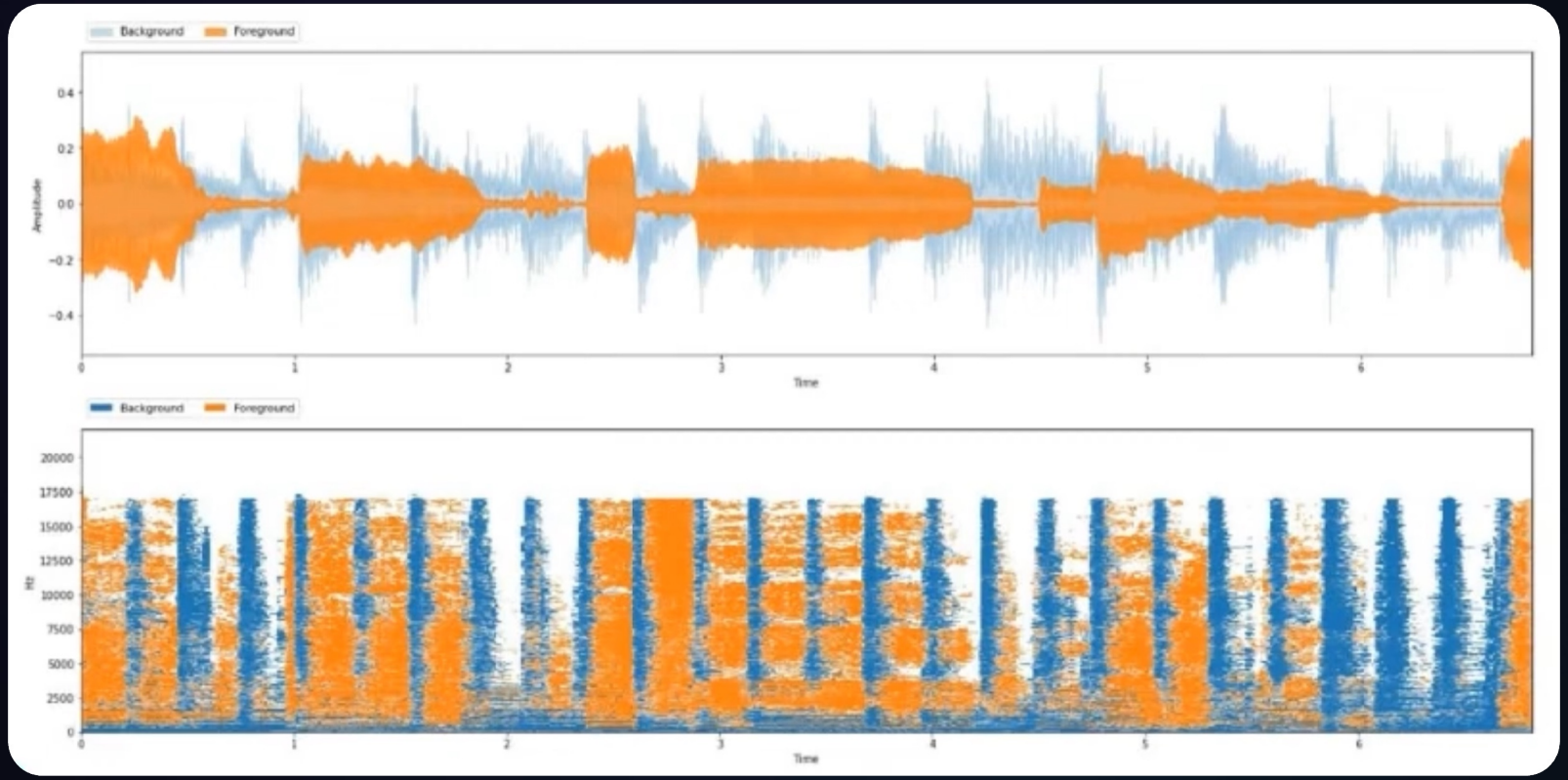


=

Source Estimate



Hybrid Approaches Win the Day



Understanding Audio Data Is Key




Vocals

Drums

Bass

Other

Accompaniment

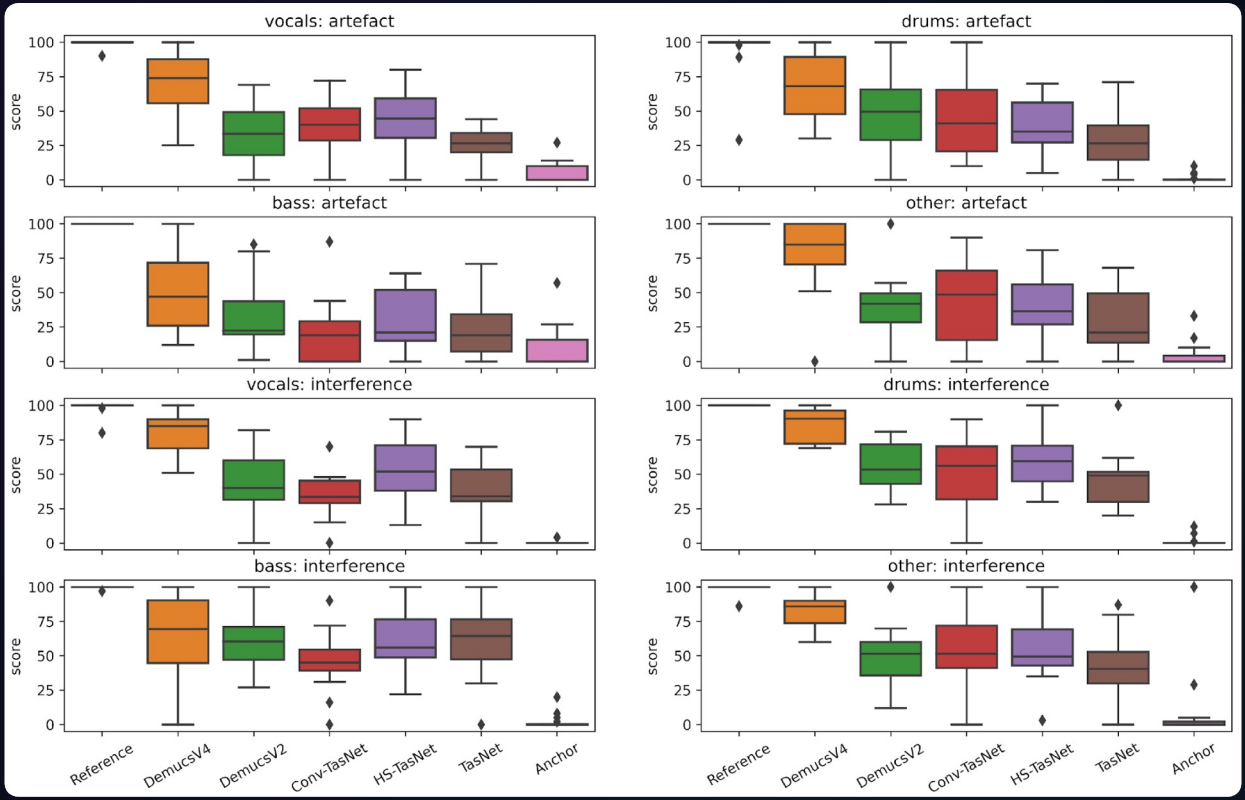
Mixture

How does it sound?

Play back from

https://l-acoustics.github.io/demix.github.io/

User Feedback



Measurements

An estimate of a Source $\hat{s}_i$, is assumed to actually be composed of four separate components, where S_{target} is the true source, and e_{interf}, e_{noise}, and e_{artif} are error terms for interference, noise, and added \text{artif, acts, respectively. The actual calculations of these terms is quite complex, so we refer the curious reader to the original paper for their exact calculation:

$$\hat{s}_i = s_{\text{target}} + e_{\text{interf}} + e_{\text{noise}} + e_{\text{artif}},$$

Using these four terms, we can define our measures. All of the measures are in terms of decibels (dB), with higher values being better. To calculate they require access to the ground truth isolated sources and are usually calculated on a signal that has been divided into short windows of a few seconds long

Measurements

Source-to-Artifact Ratio (SAR)

$$\text{SAR} := 10 \log_{10} \left(\frac{\|s_{\text{target}} + e_{\text{interf}} + e_{\text{noise}}\|^2}{\|e_{\text{artif}}\|^2} \right)$$

This is usually interpreted as the number of unwanted artifacts a source estimate has with relation to the true source

Source-to-Interference Ratio (SIR)

$$\text{SIR} := 10 \log_{10} \left(\frac{\|s_{\text{target}}\|^2}{\|e_{\text{interf}}\|^2} \right)$$

This is usually interpreted as the number of other sources that can be heard in a source estimate. This is most close to the concept of "bleed", or "leakage"

Source-to-Distortion Ratio (SDR)

$$\text{SDR} := 10 \log_{10} \left(\frac{\|s_{\text{target}}\|^2}{\|e_{\text{interf}} + e_{\text{noise}} + e_{\text{artif}}\|^2} \right)$$

SDR is usually considered to be an overall measure of how good a source sounds. If a paper only reports one number for estimated quality, it is usually SDR

Method Comparison

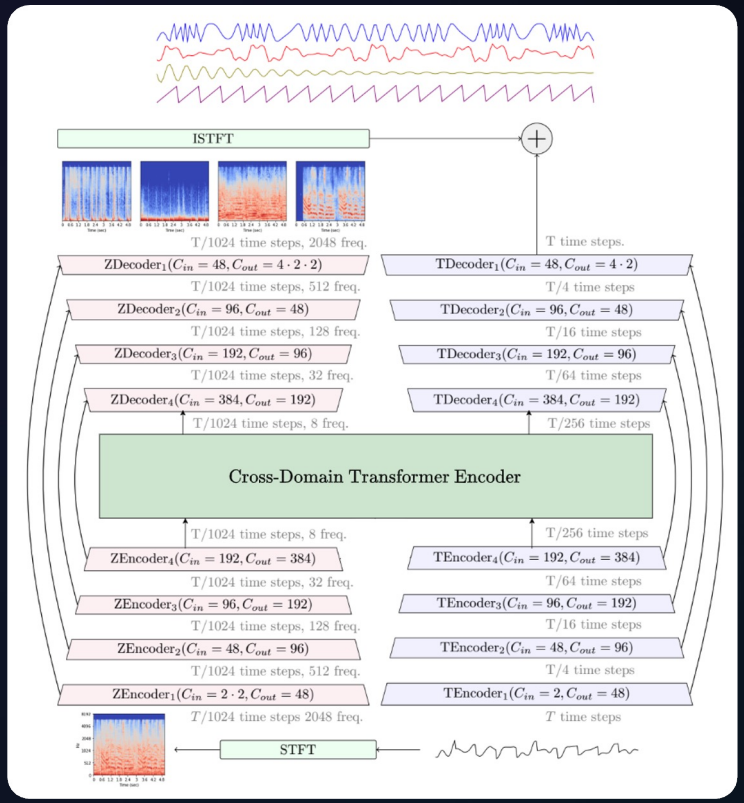
Comparing low-latency implementations of X-UMX, TasNet, and HS-TasNet to the state-of-the-art models. 'Extra' stands for how much training data in addition to MusDB is used.

	Architecture	All	Vocals	Drums	Bass	Other	Extra?	Latency (ms)	Param.	1-core (ms)	4-core (ms)	GPU (ms)
Non-causal models	Conv-TasNet [9]	5.73	6.43	6.02	6.20	4.27	✗	2,000	9 M	**52.29**	**46.50**	**16.77**
	X-UMX [16]	5.79	6.61	6.47	5.43	4.64	✗	8,000	36 M	-	-	-
	DemucsV2 [9]	6.28	6.84	6.86	7.01	4.42	✗	8,000	288 M	-	-	-
	DemucsV4 [23]	**7.52**	**7.93**	**7.94**	**8.48**	**5.72**	✗	7,800	41 M	-	-	-
Non-causal models with extra data	Conv-TasNet [9]	6.32	6.74	7.11	7.00	4.44	150	2,000	9 M	52.29	46.50	16.77
	X-UMX [32]	6.52	7.57	7.39	6.28	4.83	1505	8,000	36 M	-	-	-
	TasNet [22]	6.52	7.34	7.68	7.04	4.04	2500	15,000	29 M	**11.75**	**4.20**	**2.38**
	DemucsV2 [9]	6.79	7.29	7.58	7.60	4.69	150	8,000	288 M	-	-	-
	DemucsV4 [23]	**9.20**	**9.37**	**10.83**	**10.47**	**6.41**	800	7,800	41 M	-	-	-
Real-time Low-Latency models	X-UMX†	3.93	4.65	4.36	3.79	2.92	✗	23	31 M	9.85	4.06	**1.80**
	TasNet†	4.40	5.02	4.38	**4.73**	3.48	✗	23	51 M	9.81	4.45	1.90
	HS-TasNet†	**4.65**	5.13	**5.22**	4.59	**3.64**	✗	23	42 M	9.10	4.26	3.90
	HS-TasNet-Small†	4.48	**5.21**	4.89	4.42	3.42	✗	23	16 M	**3.98**	**1.83**	2.10
Real-time Low-Latency models with extra data	X-UMX†	4.10	4.74	4.62	3.76	3.28	150	23	31 M	9.85	4.06	**1.80**
	TasNet†	4.92	5.22	5.54	5.13	3.78	150	23	51 M	9.81	4.45	1.90
	HS-TasNet†	**5.55**	**5.97**	**6.34**	**5.62**	**4.28**	150	23	42 M	9.10	4.26	3.90
	HS-TasNet-Small†	5.01	5.51	5.75	4.93	3.86	150	23	16 M	**3.98**	**1.83**	2.10

Sources: HS-TasNet paper

Demucs v4

- Works on roughly 8s windows
- Overlap between windows
- Offline!



Source: Hybrid Transformers for Music Source Separation, Simon Rouard, Francisco Massa, and Alexandre Defossez,ICASSP 23

HS TasNet

- Works on roughly 23ms windows
- Reasonably realtime

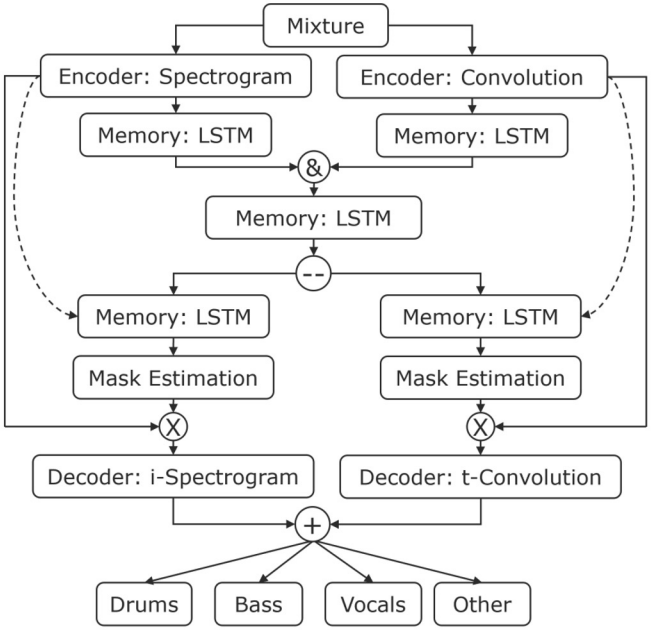


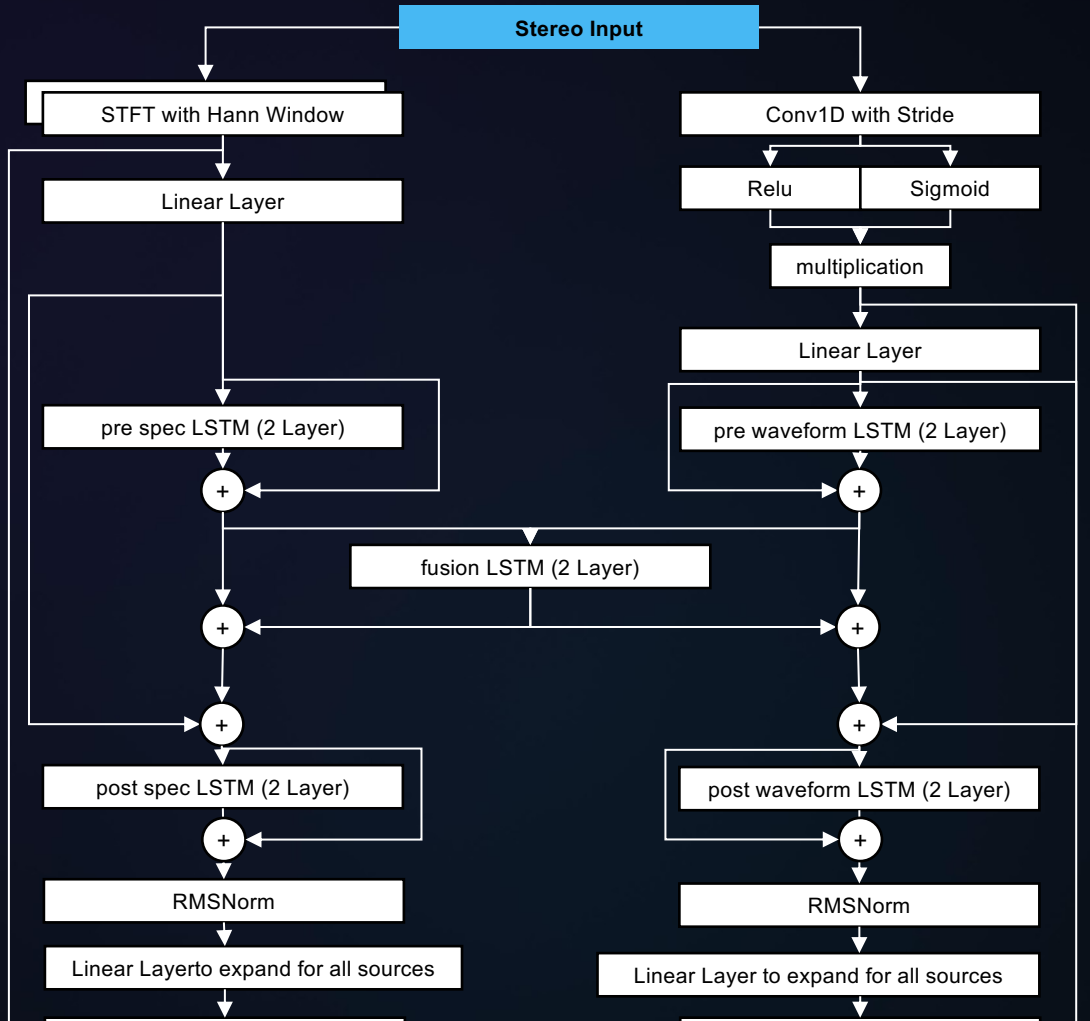
Fig. 1. Structure of the HS-TasNet. '&', '--', '+', 'i-', 't-', and dotted lines stand for concatenate, split, sum, inverse, transpose and skip connections respectively.

Source: Real-time Low-latency Music Source Separation using Hybrid Spectrogram-TasNet
Satvik Venkatesh, Arthur Benilov, Philip Coleman, Frederic Roskam

Lucidrains implementation

Structure is slightly adapted

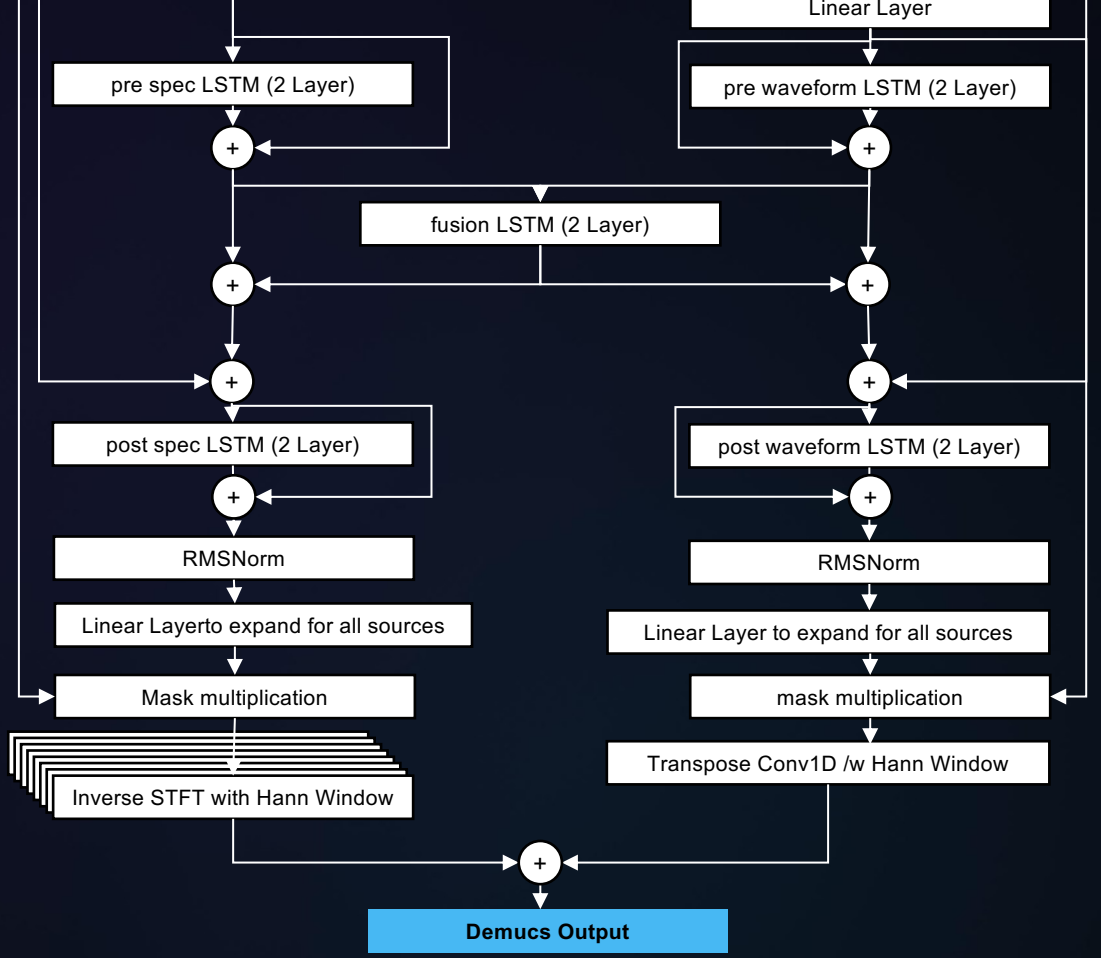
- Every LSTM cell is setup as a ResNet architecture
- The Spectrogram branch uses real and complex values
- Skip connections are ResNet style connections
- An RMSNorm cell in each branch is present (improves training stability) according to the authors



Lucidrains implementation

Structure is slightly adapted

- Every LSTM cell is setup as a ResNet architecture
- The Spectrogram branch uses real and complex values
- Skip connections are ResNet style connections
- An RMSNorm cell in each branch is present (improves training stability) according to the authors



Performance Numbers

Performance with STFT 1024 and 512 overlap, 44.1kHz, buffer size 512

Real-time Ratio =

real-time / processing time

=> >1 is real-time

Pytorch (GPU)

- RTR 0.58

CPU (with Eigen)

- RTR 0.18125

GPU audio

- RTR 6.955
- 16 instance RTR 1.078 – can run 16 instances in parallel

Model Details

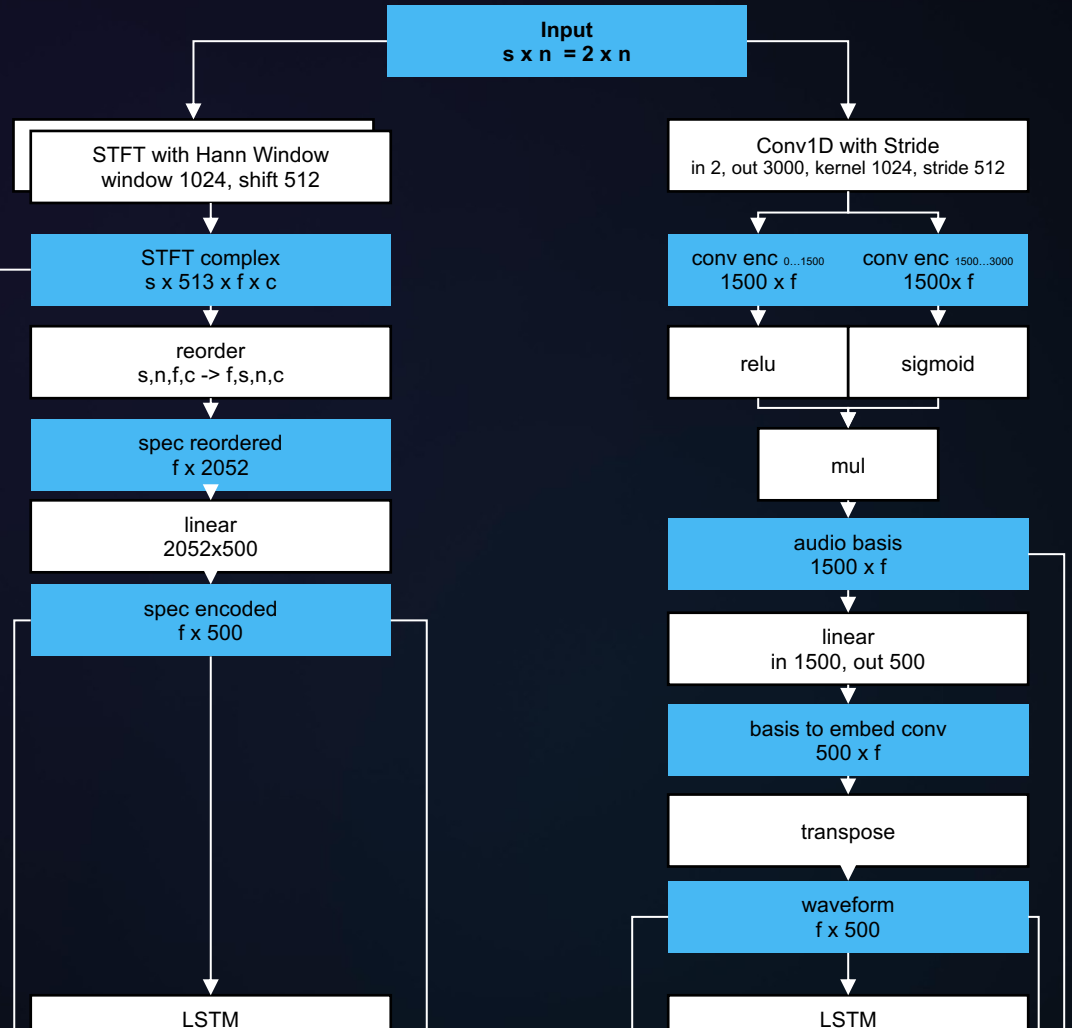
Full model details

Exported from PyTorch

n = samples
s = channels = 2
c = complex dim = 2
f = frames = 1 + (n - fft_size) / hop_length

Tensor

Processing Block



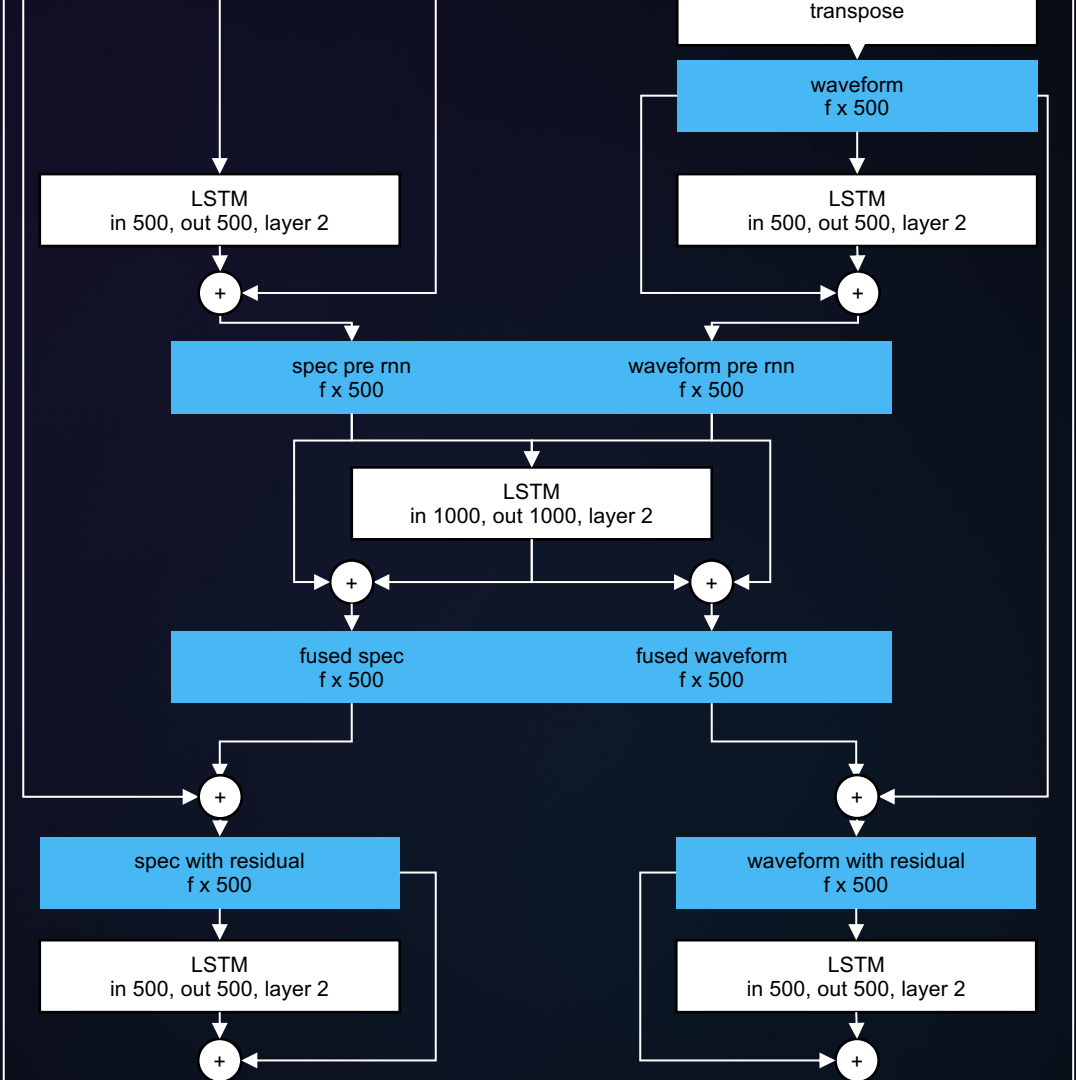
Full model details

Exported from PyTorch

n = samples
s = channels = 2
c = complex dim = 2
f = frames = 1 + (n - fft_size) / hop_length

Tensor

Processing Block



Full model details

Exported from PyTorch

n = samples
s = channels = 2
c = complex dim = 2
f = frames = 1 + (n - fft_size) / hop_length

Tensor

Processing Block

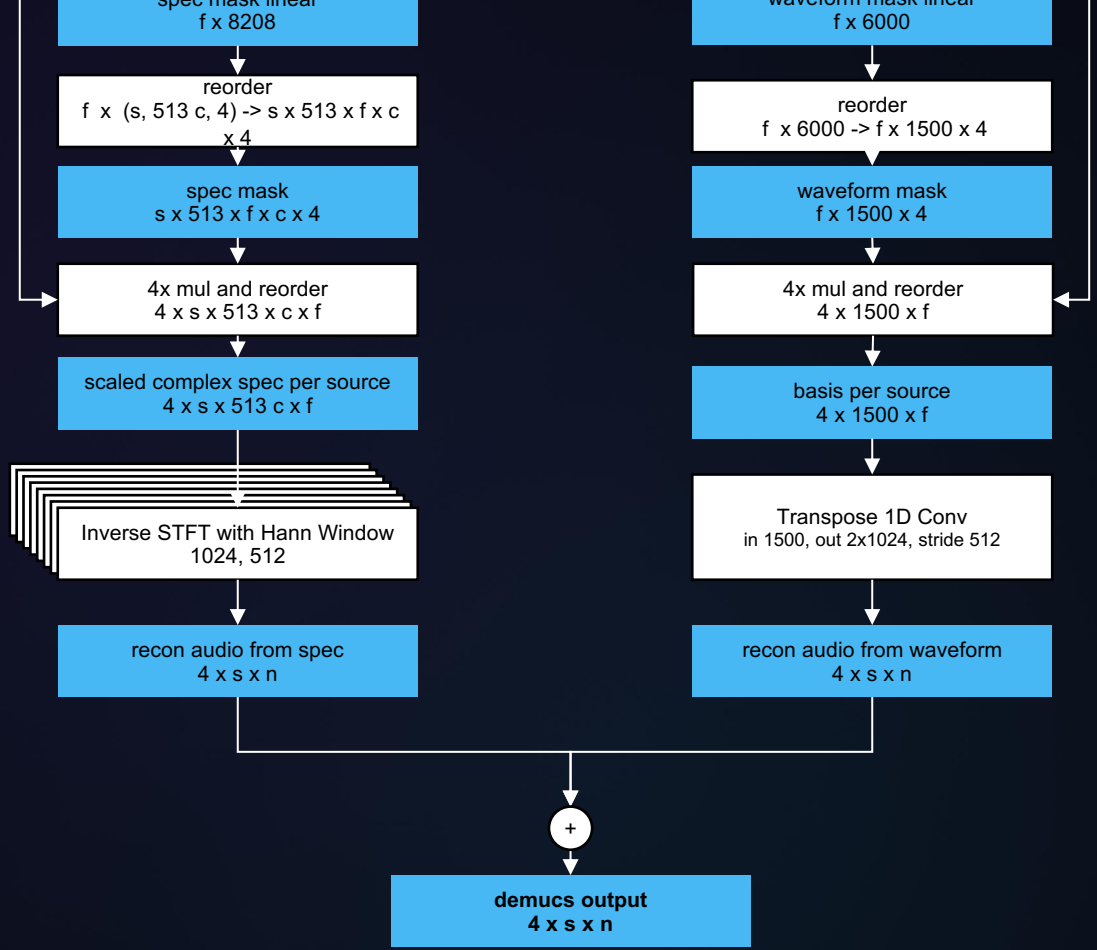
Full model details

Exported from PyTorch

n = samples
s = channels = 2
c = complex dim = 2
f = frames = 1 + (n - fft_size) / hop_length

Tensor

Processing Block



Have a Look at Pytorch

Unfortunately export with torch.onnx.export failed

- We want a bit more control here anyway, the pytorch implementation supports longer input audio and multiple batches
- It actually does not perform well for short buffer lengths (normalization of the Hann Window, more about it later on)

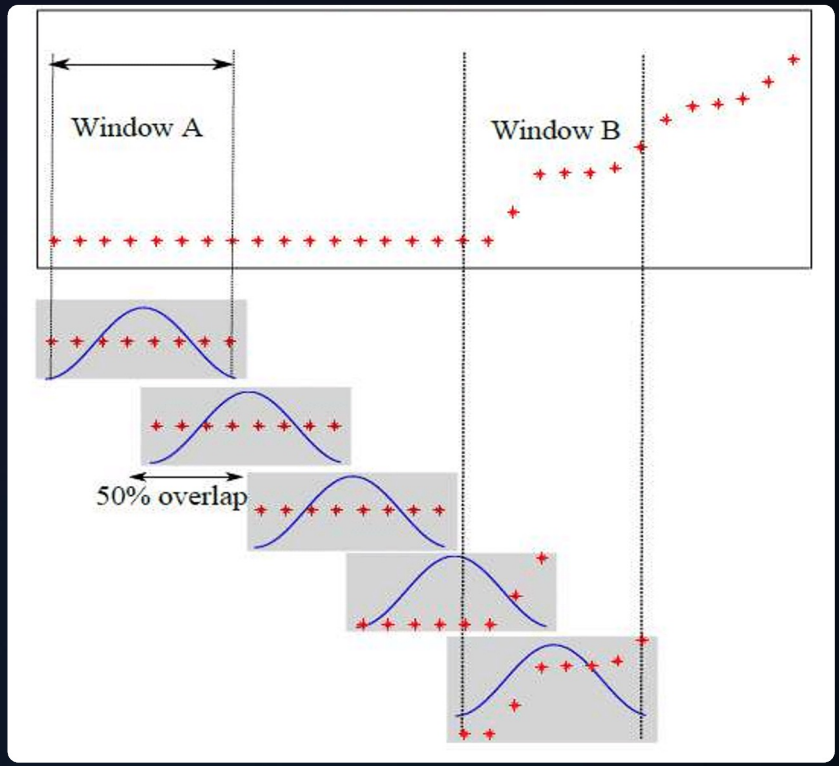
What Is Our Goal?

Low actual latency

- We want to work on the smallest possible buffer length!
- Answer: 512 – the stride used in both the spectral and waveform branch
- Note: we need 1024 samples to process, but due to the overlap every 512 samples gives us a new 1024 samples buffer
- So we need to keep the last 512 samples around

Get rid of all unnecessary reordering etc

- We have full control over memory flow



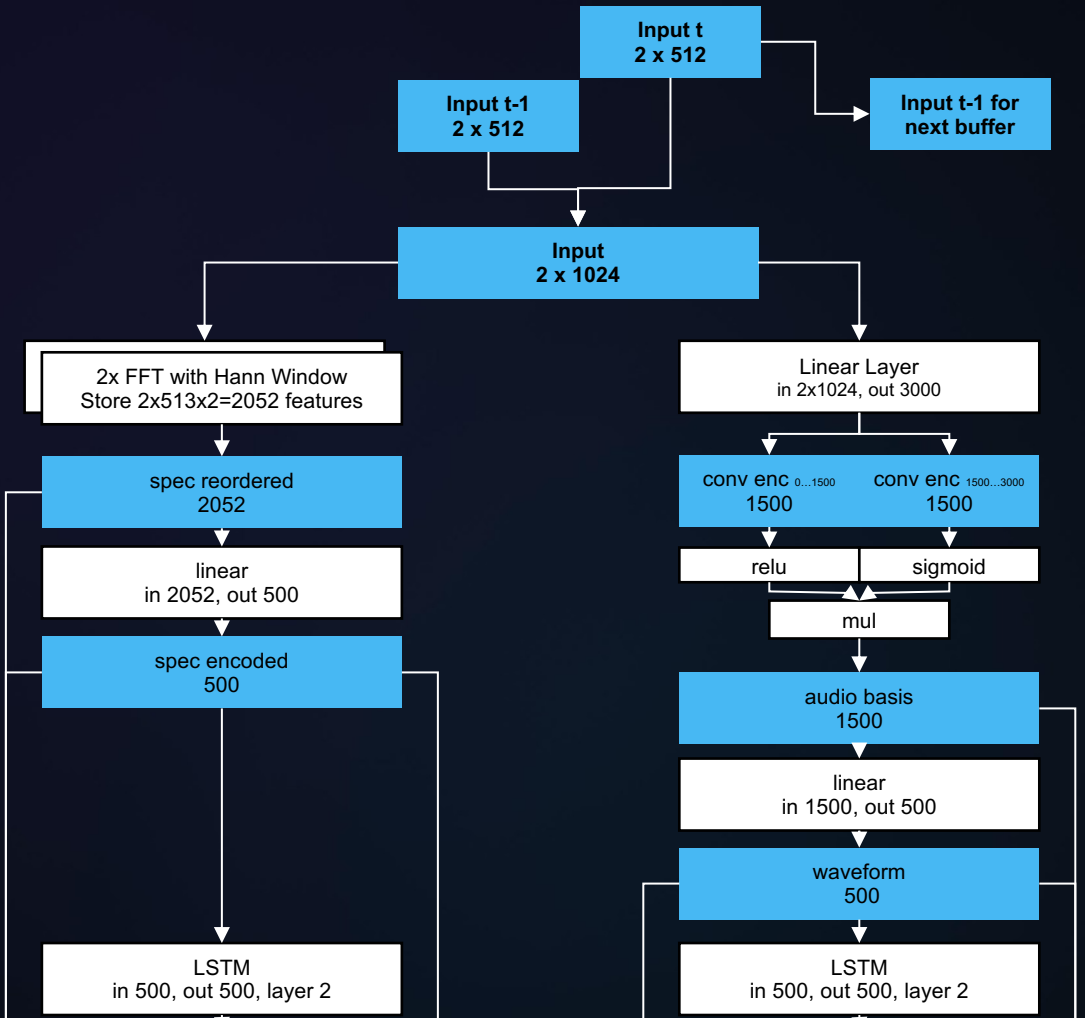
Streamlined Single

n = samples
s = channels = 2

c = complex dim = 2
f = frames = 1 + (n - fft_size) / hop_length

Tensor

Processing Block



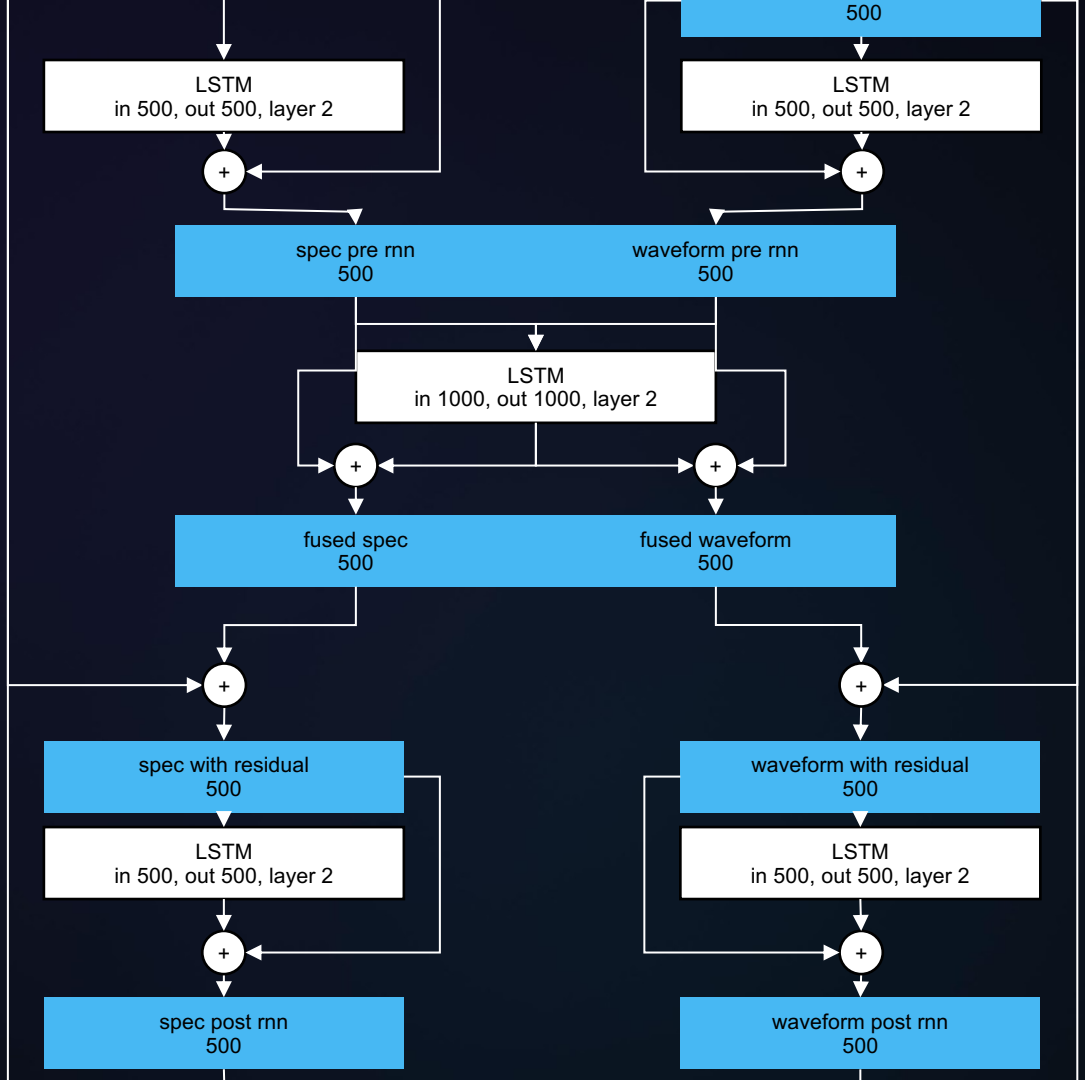
Streamlined Single

n = samples
s = channels = 2

c = complex dim = 2
f = frames = 1 + (n - fft_size) / hop_length

Tensor

Processing Block



Streamlined Single

n = samples
s = channels = 2

c = complex dim = 2
f = frames = 1 + (n - fft_size) / hop_length

Tensor

Processing Block

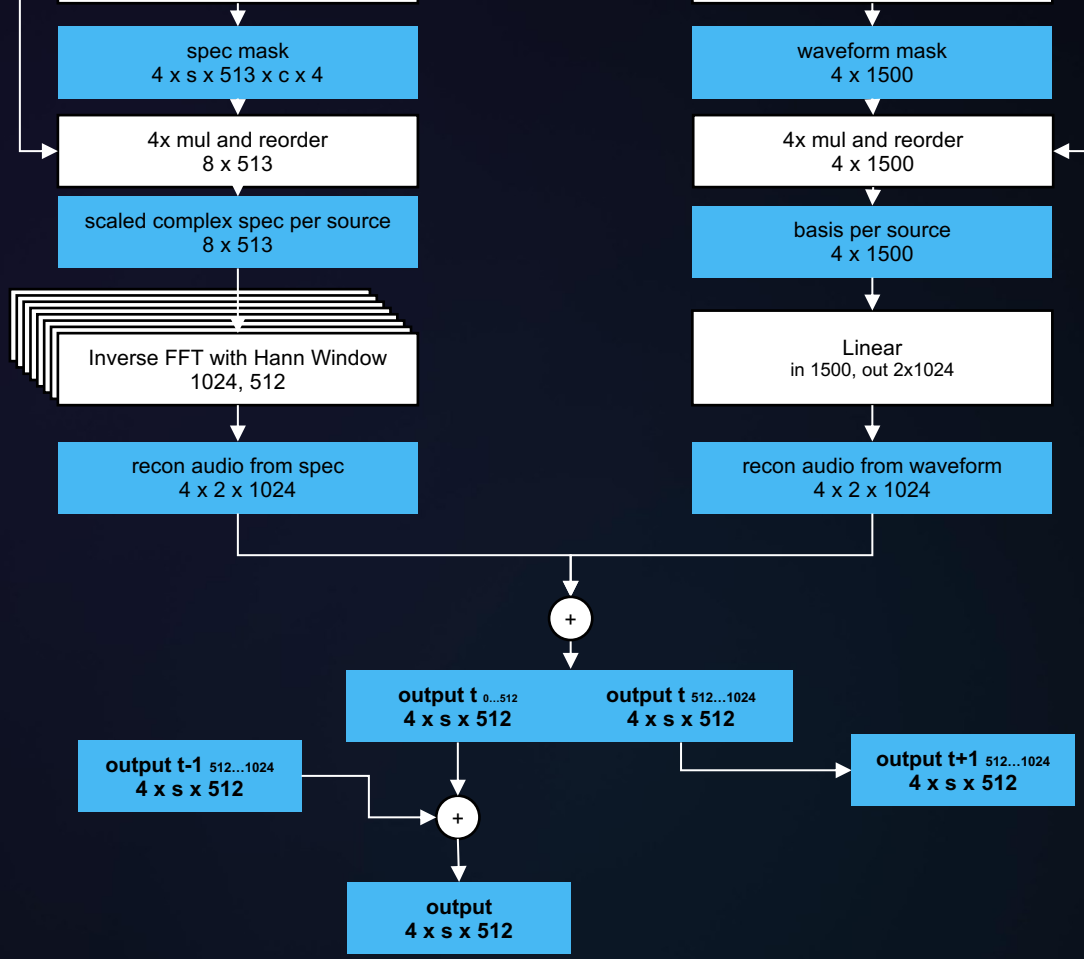
Streamlined Single

n = samples
s = channels = 2

c = complex dim = 2
f = frames = 1 + (n - fft_size) / hop_length

Tensor

Processing Block



Building Blocks Needed

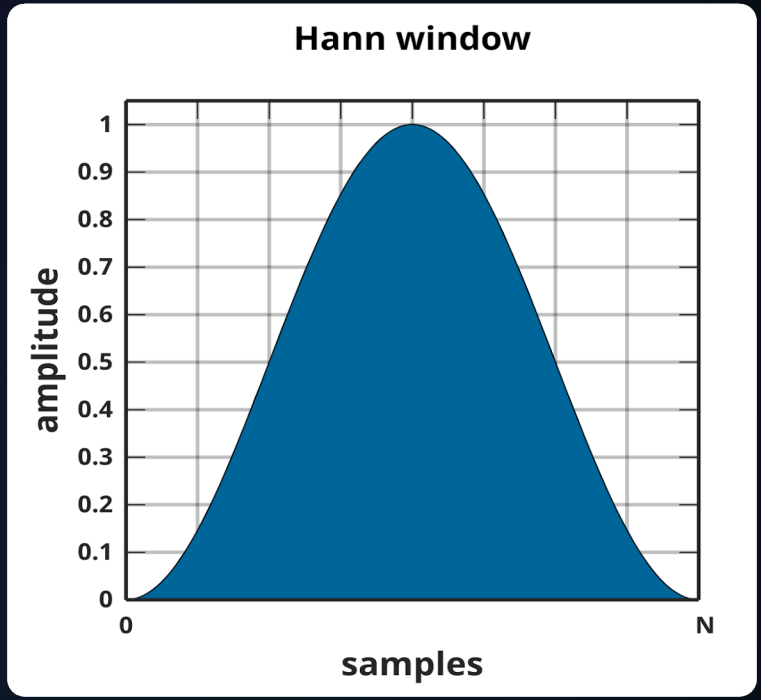
Fourier Transform	Neural Network	Activation Functions	Math
FFT with Hann Window	Linear Layer	Relu	Add
Inverse FFT with Hann Window	LSTM	Sigmoid	Mul
	RMSNorm		

FFT with Hann Window

- Real to complex FFT with size 1024
- Fourier domain signal is conjugate-symmetric, we only need 513 complex values
- DC and Nyquist bins are real (actually 512 would complex values would be sufficient, but we stuck to the pytorch format)
- Hann Window is a simple scaling factor on the input

$$w[n] = \sin^2\left(\frac{\pi n}{1024}\right), \qquad \text{if } n \geq 0, n < 1024$$

- No scaling is applied in the Lucidrains implementation here



Inverse FFT with Hann Window

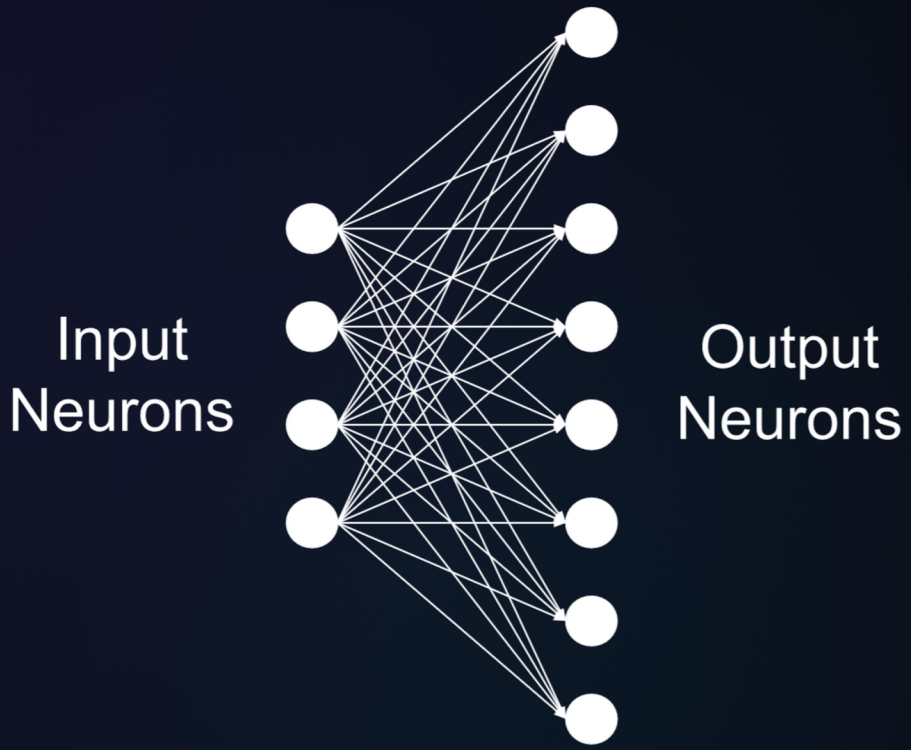
- Complex to real inverse FFT with size 1024
- 513 complex values turned into 1024 signal samples
- Normalization is applied here: 1/1024
- Hann Window is again applied on the output
- Hann Window normalization is carried out to rescale output according to contribution

$$w_{corr}[n] = \frac{1}{w^2[n-512] + w^2[n] + w^2[n+512]}$$

Linear Layer

y = Ax + b

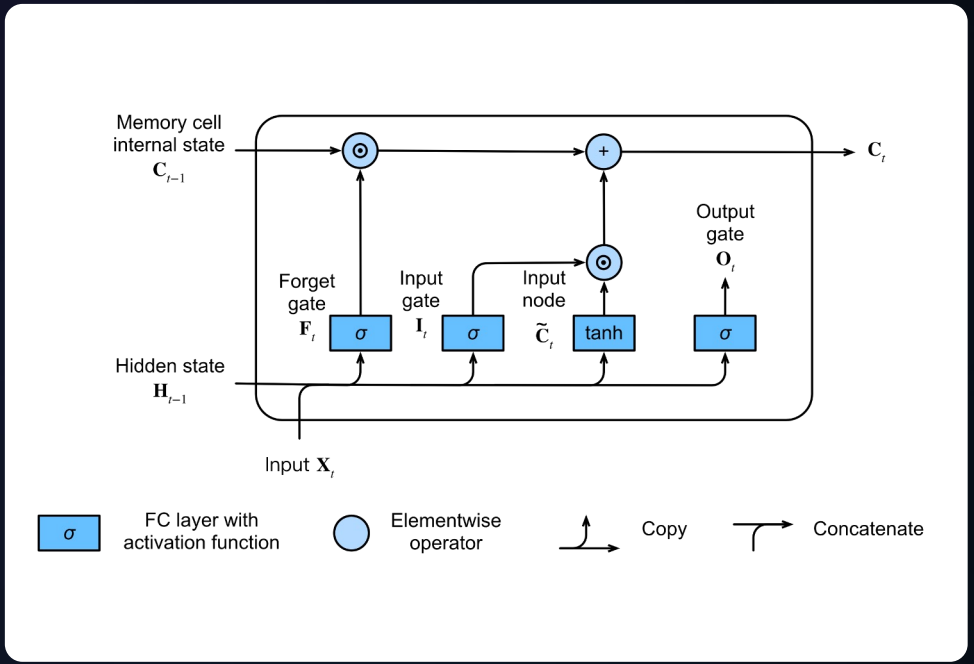
- ***A*** weight matrix
- ***b*** bias
- ***x*** input vector
- ***y*** output vector



LSTM: Multi-layer Long Short-term Memory (LSTM) RNN

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

- Memory module to keep information around
- Weights and biases of all 4 branches are typically combined into one large matrix and vector, and the results then separated
- The combination operations require 4 elements of the large matrix vector product



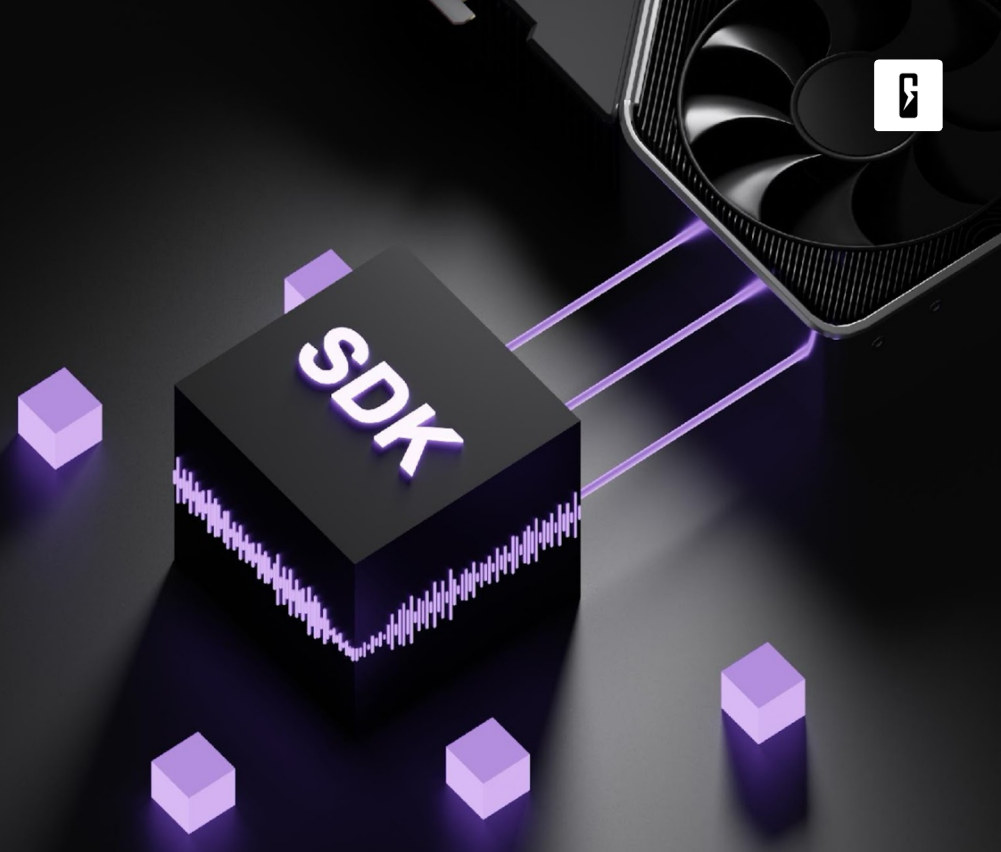
RMSNorm

$$y_i = \frac{x_i}{RMS(x)} \cdot \gamma_i \text{ with } \qquad RMS(x) = \sqrt{\epsilon + \frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

Trivial operation, requires a reduction over all elements in the input vector

GPU Audio SDK Overview

- Cross-platform
- Many layers that can be used as desired
- Low latency
- High performance DSP and ML



GPU AUDIO SDK Workflow Schematics

Engine API

External-facing API used to initialize, configure, and launch audio processing from your app or host

GPU Audio Component

Core engine that runs the audio graph, manages execution, and handles communication with the GPU

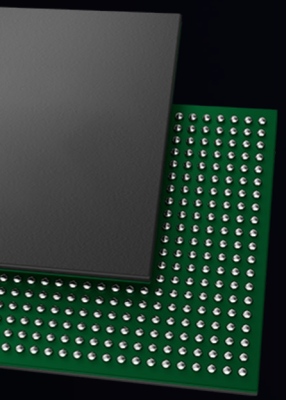
Processor API

- Implements custom audio effects and processors
- Supports GPU code integration, memory setup, and scheduler configuration

DSP Components Library

Collection of modular building blocks: FFT, convolution, NNs, and more

SDK Cross-Platform Capabilities

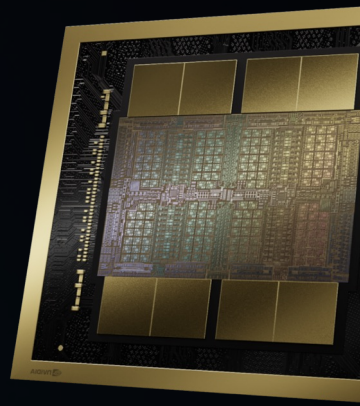


Unified CPU/DSP-side interfaces

- Initialization
- Compute Graph Setup
- Port Management
- Memory Management
- Parameter passing

Common device-side C++ style language*

- Syncthreads, shared memory, warp communication, etc
- Cache memory operations
- Thread management



Write your code once, and watch as it automatically compiles and deploys seamlessly across multiple platforms

Processor Launcher: Entities

GPU Audio

The entry point for launching processors — accessed via the Engine API

Graph Launcher

The component that triggers processor execution on the GPU

Processing Graph

Connects multiple processors into a single audio flow

Processor

Individual audio effect implementing the Processor API

Processor API quick info

```
// dynamic library interface
ErrorCode
CreateModuleInfoProvider_v2(...);
ErrorCode
DeleteModuleInfoProvider_v2(..);
```

Functions for providing information about the supported platforms

```
ErrorCode
CreateDeviceCodeProvider_v2(...);
ErrorCode
DeleteDeviceCodeProvider_v2(...);
```

Functions for providing the GPU code for a specific platform

```
ErrorCode CreateModule_v2(...);
ErrorCode DeleteModule_v2(...);
```

Functions for providing the GPU code for a specific platform

```
class DeviceCodeProvider {
public:
  ErrorCode GetDeviceCode(...);
};
```

Simple method to get the precompiled binary code for GPU execution. Compilation and setup taken care of by our build environment.

```
class Module {
public:
  ErrorCode CreateProcessor(...);
  ErrorCode DeleteProcessor(...);
};
```

Methods for creating a processor; typically, just new/delete on custom Processor class

```
class ModuleInfoProvider {
public:
  ErrorCode GetSupportPlatformInfo(...);
  ErrorCode GetModuleInfo(...);
  ErrorCode GetProcessorExecutionInfo(...);
};
```

Methods to to get information about the supported platforms, module's version, and the GPU code entry functions. Most of them can be auto generated from simple meta data

Processor API quick info

```
class Processor {
public:

 ErrorCode SetData(...);
 ErrorCode GetData(...);
 ErrorCode GetInputPort(...) ;

 ErrorCode OnBlueprintRebuild(...);

 ErrorCode PrepareForProcess(...);
 ErrorCode PrepareChunk(...);

 void OnProcessingEnd(...);
}
```

Main interface to implement when creating your own processor

— Methods for passing custom parameters to processors (simple pass through)

— Method to connect input data to the processor and connect from other processors (graph)

— Method to provide information about which functions to execute on the GPU

Preparation function for reacting to new input data and providing parameters for GPU execution

— Optional callback for when processing on the GPU is completed.

```
class MemoryManager{
public:
 GpuMemoryPointer AllocateGpuMemory(...);

CpuMemoryPointerAllocatePinnedCpuMemory(.
..);

 void MemCpyCpuToGpu(...) ;
 void MemCpyCpuToGpu(...);

 Future MemCpyCpuToGpuAsync(...);
 Future MemCpyCpuToGpuAsync(...);
}
```

— Provided to each new processor for platform independent memory management.

```
class PortFactory{
public:
 OutputPortPointer
CreateDataPort(...);
}
```

Provided to each new processor for generating output ports that can be used to connect to other processors or output buffers back to the DAW (or other destinations).

Device Execution Quick Info

```
class DeviceProcessor {
public:
  template <class Context>
__device_fct void init(Context context,
unsigned int bufferLength) __device_addr;
  template <class Context>
__device_fct void my_process(Context
context, __device_addr ProcParam* params,
__device_addr TaskParam* task_params,
__device_addr float* __device_addr*
input, __device_addr float*
__device_addr* output) __device_addr;
};
```

- Every GPU processor only needs an init method. And can have an arbitrary number of process functions (name does not matter)
- Keywords to annotate functions and pointers (needed for MAC compilation)
  - __device_fct … a function on the GPU
  - __device_addr … a pointer to GPU memory (also needed for member functions of device memory objects)
  - __threadgroup_addr … a pointer to shared memory
  - __thread_addr … a pointer to a local variable
- The Context class abstracts all platform dependent GPU code (thread id, synchronization, shfl, shared memory etc).You typically want to pass the context into all functions you call.
- Every process method has the following additional parameters:
  - ProcParam* params … custom parameter passed to all process methods of the processor
  - TaskParam* task_params … specific parameters for individual process methods (in this case there is only one)
  - float** input … input port data (one pointer for each input port)
  - float** output … output port data (one pointer for each output port)

```
// final declaration of the processor (in a cu file)
DeclareProcessorStep(DeviceProcessor, 0, my_process_0,
float, ProcParam, TaskParam0);
DeclareProcessorStep(DeviceProcessor, 1, my_process_1,
float, ProcParam, TaskParam1);
DeclareProcessor(DeviceProcessor, 2);
```

- Each process method needs to be declared (and numbered). The input data type (float) and the parameter types need to be specified.
- The final processor declaration only need to class name and the number of process functions (2 in this case)

Jupyter Environment

Hands-on demo in Jupyter environment

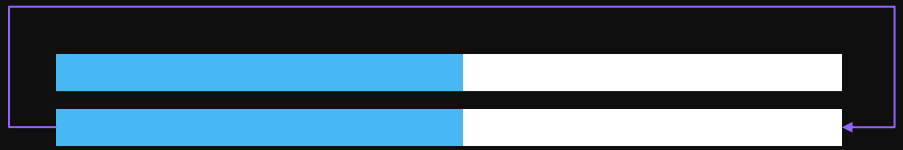
Sign up credentials:

https://workshop.gpu.audio

Username: workshop
Password: 8uWpaR36zwUXWDBcg4eeZGK5

GPUA Implementation:
GPU-Building Blocks

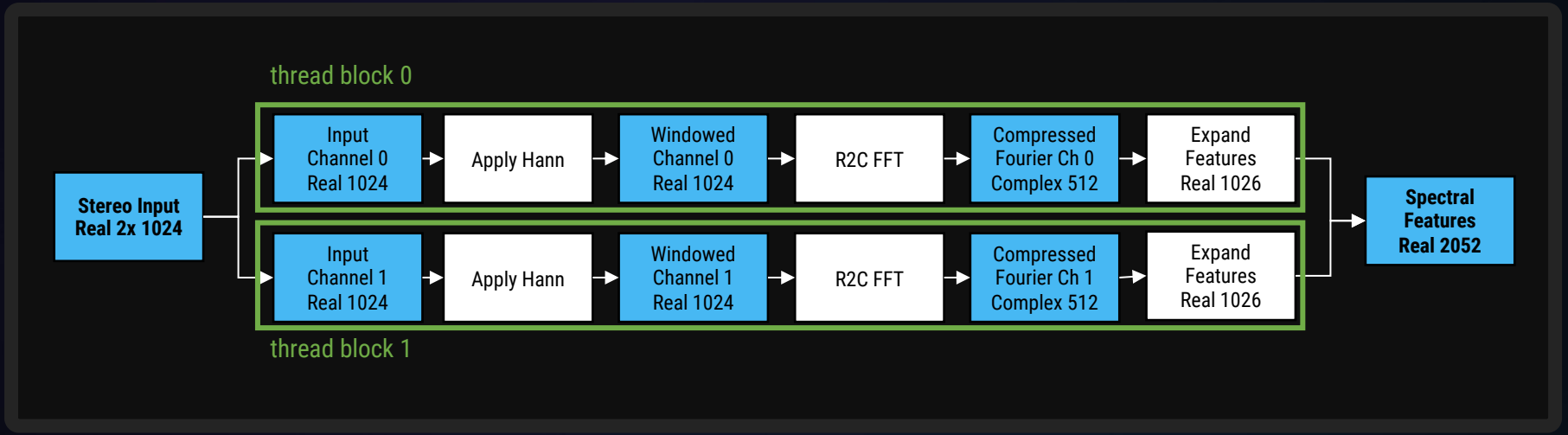
Multichannel Ringbuffer



- Often used for delay lines
- Size should be chosen such that sufficient history can loaded
- Here we only need to keep the last 512 samples around
- Could be used for longer window sizes of short input buffers for accumulation
- Cursor to capture current position
- Load and Store for elements, vectors, matrix or from buffer pointer
- Access iterator

```
template <uint32_t CHANNELS, uint32_t
RINGBUFFER_SIZE, typename TYPE>
class MultiChannelRingBuffer {
  public:

  template <class Context>
  void LoadChannelData(
     Context& context,
     uint32_t channel, uint32_t cursor,
     uint32_t num, TYPE* data) const;

  template <class Context>
  void StoreChannelData(
     Context& context,
     uint32_t channel, uint32_t cursor,
     uint32_t num, const TYPE* data);

  LinearChannelAccess
  GetLinearChannelAccess(
     uint32_t channel, uint32_t cursor);
};
```

FFT with Hann Window

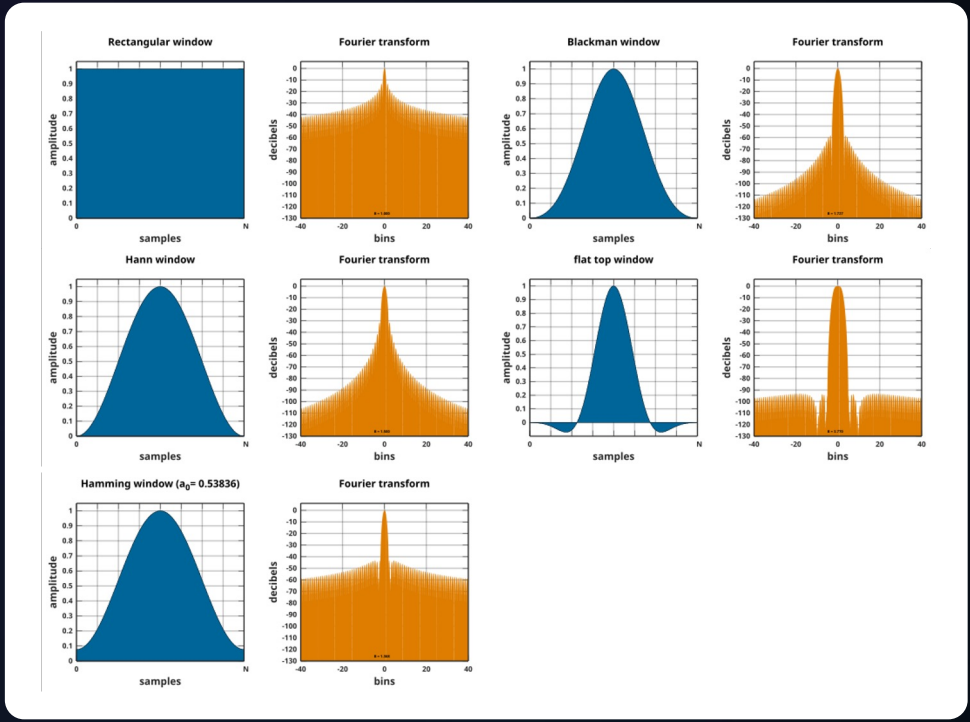


- Each channel can be worked on by a separate thread block
- Filter functions provided by our SDK
- Real-to-Complex FFT provided by our SDK
- Only need to move memory such that it matches the Lucidrains format

Window Function

- Various window functions provided
- Only correct for 0 … length
- Called on an individual thread basis

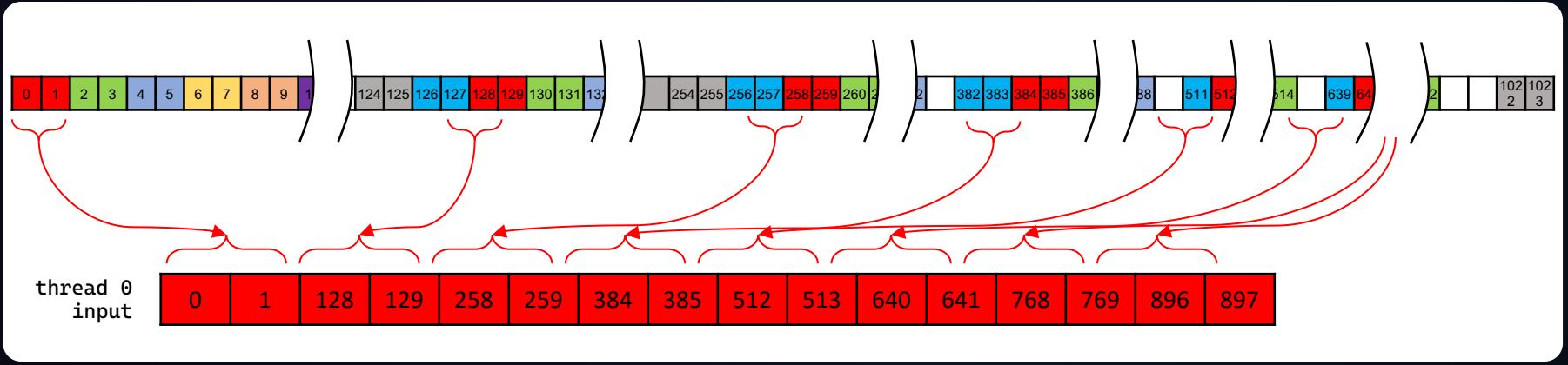
```
float apply_hann(uint32_t x, uint32_t
length);

float apply_hamming(uint32_t x, uint32_t
length);

float apply_blackman_harris(uint32_t x,
uint32_t length);

float apply_flattop(uint32_t x, uint32_t
length);

float apply_rectangle(uint32_t x, uint32_t
length);
```



Real-to-Complex FFT

- Input and output pointers to thread local array
- Every thread holds multiple elements for efficiency reasons
  for today: 8 float2 elements – block size = (1024/2) / 8 = 64
- Twiddle factors as lookup table for efficiency
  provided by the SDK
- Each thread provided pairs of elements 64 (block size) float2 apart

```
void
block_fft_r2c_<FFTSIZE>_<TUNINGPARAMETERS>(
    uint32_t tid, float* smem,
    float2* input, float2* output,
    const float2* twiddle_LUT,
    const float2* twiddle_LUT_inverse);
```



Real-to-Complex FFT

```
void
block_fft_r2c_<FFTSIZE>_<TUNINGPARAMET
ERS>(
   uint32_t tid, float* smem,
   float2* input, float2* output,
   const float2* twiddle_LUT,
   const float2* twiddle_LUT_inverse);
```

Output comes as complex numbers, whereas the first thread gets DC and Nyquist frequencies

DC	Ny	Re[1]	Im[1]	Re[2]	Im[2]	Re[3]	Im[3]	Re[4]	Im[4]	Re[5]	Im[5]	•••	Re[510]	Im[510]	Re[511]	Im[511]

thread 0 output

DC	Ny	Re[64]	Im[64]	Re[128]	Im[128]	Re[192]	Im[192]	Re[256]	Im[256]	Re[320]	Im[320]	Re[384]	Im[384]	Re[448]	Im[448]

thread 1 output

Re[1]	Im[1]	Re[65]	Im[65]	Re[129]	Im[129]	Re[193]	Im[193]	Re[257]	Im[257]	Re[321]	Im[321]	Re[385]	Im[385]	Re[449]	Im[449]

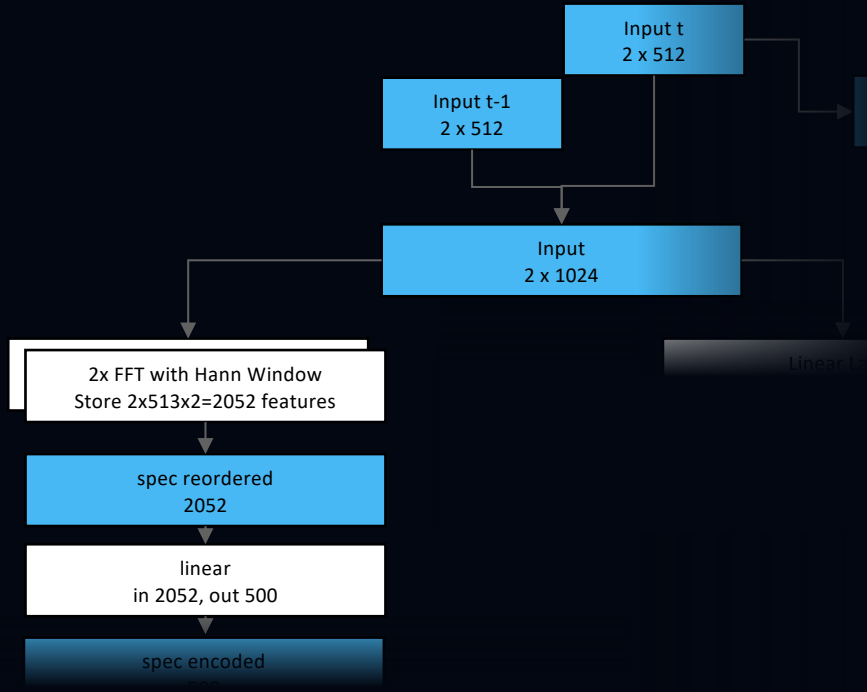
⋮

Output for Lucidrains

DC	0	Re[1]	Im[1]	Re[2]	Im[2]	Re[3]	Im[3]	Re[4]	Im[4]	Re[5]	Im[5]	•••	Re[510]	Im[510]	Re[511]	Im[511]	Ny	0

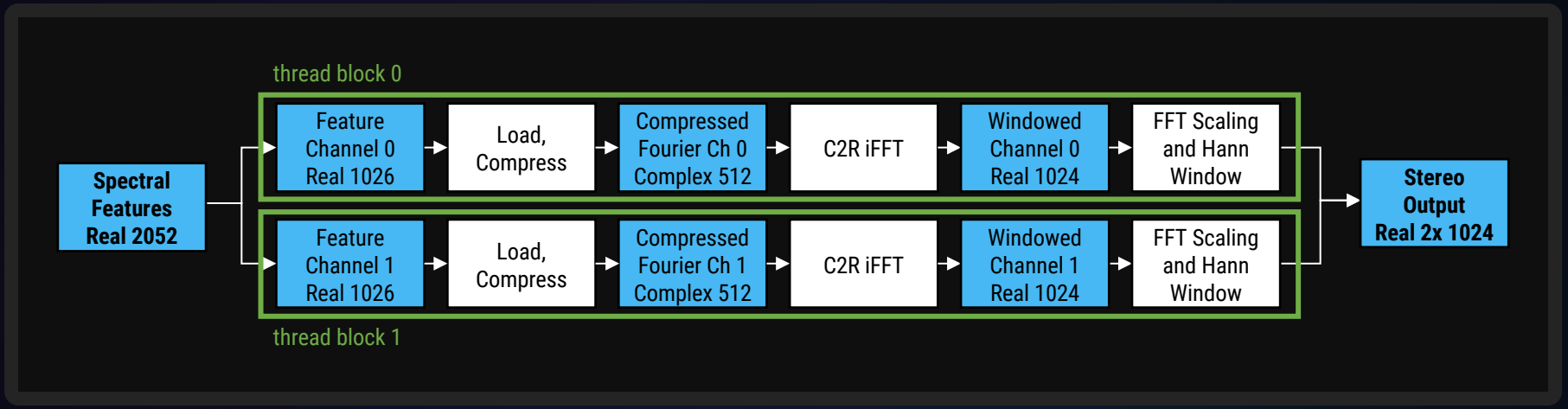
Try it out now

Write the STFT



```
template  typename Context  typename T>
static __device_fct __forceinline_fct void stft(__thread_addr Context& context,
              __threadgroup_addr float* smem,
              uint32_t channel_id,
              __device_addr T const& input,
              __device_addr float* enc_output,
              uint32_t signal_length,
              uint32_t hop_size,
              __device_addr float2 const* twiddle_LUT, __device_addr float2 const* twiddle_inverse_LUT) {

              // note that input will be LinearChannelAccess
              uint32_t tid = context.threadId();

        float2 input_data[8];
        for (uint32_t i = 0; i < 8; ++i) {
              // load data and apply hann window

              };

        float2 output_data[8];
        linear_block_fft_r2c_512_8_8_8(tid, smem, input_data, output_data, twiddle_LUT, twiddle_inverse_LUT);

       // channel > bins > real/imag
              for (uint32_t i = 0; i < 8; ++i) {
              // write the complete feature vector to memory according to   channel > bins > r/c
              // special handling for DC and Nyquist frequency

              }
}
```

Inverse FFT with Hann Window



- Again each channel can be worked on separately again
- Again reorganize memory from Lucidrain format to compressed building block
- Complex to real inverse FFT exactly inverse to forward FFT
- Normalization with $1/1024$
- Hann window applied again and normalization according to

$$w_{corr}[n] = \frac{1}{w^2[n-512]+w^2[n]+w^2[n+512]}$$

Complex-to-Real iFFT

- Input and output pointers to thread local array
- Every thread holds the complex elements of the compressed

- Only first half of the data as second is only the complex conjugate

- Thread 0 holds both the DC and Nyquist frequencies

- Twiddle factors the same as with the real to complex FFT
- No normalization applied

```
void
block_fft_r2c_<FFTSIZE>_<TUNINGPARAMETERS>
(
    uint32_t tid, float* smem,
    float2* input, float2* output,
    const float2* twiddle_LUT,
    const float2* twiddle_LUT_inverse);
```

thread 0 input

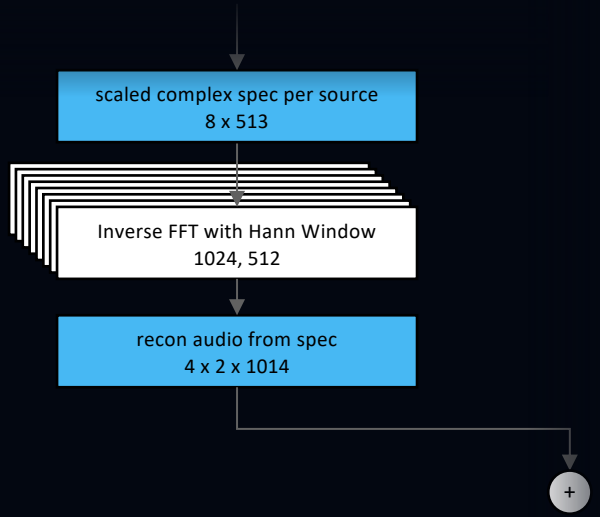
DC	Ny	Re[64]	Im[64]	Re[128]	Im[128]	Re[192]	Im[192]	Re[256]	Im[256]	Re[320]	Im[320]	Re[384]	Im[384]	Re[448]	Im[448]

thread 1 input

Re[1]	Im[1]	Re[65]	Im[65]	Re[129]	Im[129]	Re[193]	Im[193]	Re[257]	Im[257]	Re[321]	Im[321]	Re[385]	Im[385]	Re[449]	Im[449]

Try it out now

Write the inverse STFT



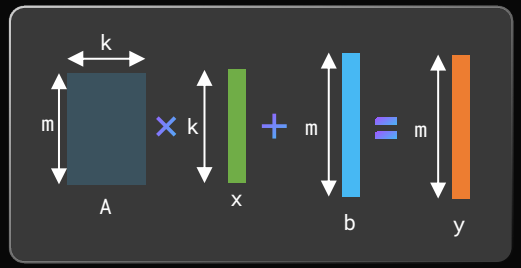
```
template  typename Context, typename  >
static __device_fct __forceinline_fct void inv_stft(__thread_addr Context& context,
             __threadgroup_addr float* smem,
             uint32_t channel_id,
             __device_addr float  const real,
             __device_addr float  const imag,
             __device_addr T& output,
             uint32_t signal_length,
             uint32_t hop_size,
             __device_addr float2 const* twiddle_LUT, __device_addr float2 const* twiddle_inverse_LUT) {

             // note that output will be LinearChannelAccess
             uint32_t tid = context.threadId();

             float2 data[8];
        for (uint32_t i = 0; i < 8; ++i) {
             // load data and combine dc and Nyquist bins to thread 0, data[0]


             };

             linear_block_fft_c2r_512_8_8_8 (tid, smem, data, data, twiddle_LUT, twiddle_inverse_LUT);

             for (uint32_t i = 0; i < 8; ++i) {
             // apply 1/1024 scaling

             // apply Hann window

             // normalize over some of window contributions at this location hann^2

             // write output


             }
}
```

Linear Layer

Simple Matrix Vector multiplication

$$y = Ax + b$$

- A weight matrix
- b bias
- x input vector
- y output vector




```
template <uint32_t ROWS, uint32_t COLS,
typename TYPE, MatrixStorage STORAGE>
struct Matrix;

template <uint32_t ELEMENTS,
typename TYPE = float>
struct Vector;

template <typename Context, …>
void MultiplyAddBias(Context& context,
          Y_TYPE& C,
          A_TYPE const& A,
          X_TYPE const& x,
          BIAS_TYPE const& Bias,
          uint32_t blocks);

template <typename Context, …>
void Multiply(Context& context,
          Y_TYPE& C,
          A_TYPE const& A,
          X_TYPE const& x,
          uint32_t blocks);
```

- Matrix and Vector classes for access from CPU and GPU
- Support for distribution of multiplication across multiple blocks
- Support for larger matrices and higher performance

Activation Functions

- **Relu - Rectified linear unit**

$$ReLU(x) = \max(0, x)$$

- **TanH - hyperbolic tangent**

a) $tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{2x}-1}{e^{2x}+1}$

b) Fast version without exponential from our SDK

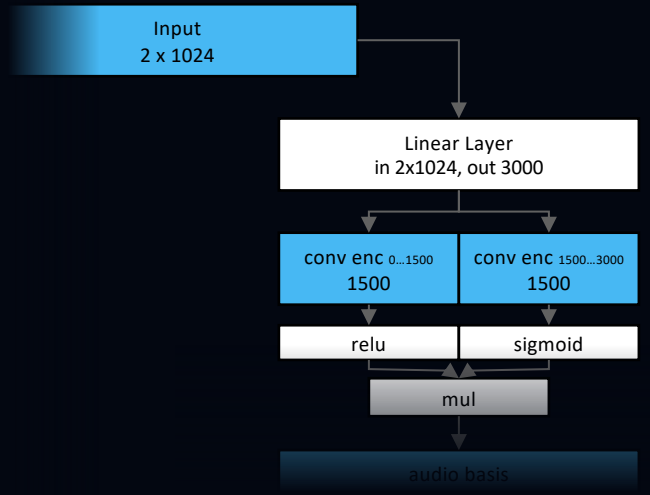
- **Sigmoid**

a) Logistic function (as used by pytorch): $\sigma(x) = \frac{1}{1+e^{-x}}$

b) Fast version when using fast tanh: $\sigma_{fast}(x) = \frac{1}{2}\tanh_{\text{fast}}\left(\frac{x}{2}\right) + \frac{1}{2}$

Try it out now

Write a simple matrix vector multiplication



```
 template <typename Context>
__device_fct void conv_encode_task(Context context,
              __device_addr ProcessorParameter* processor_param,
              __device_addr ConvEncodeParams* task_param,
              __device_addr float* __device_addr* input,
              __device_addr float* __device_addr* output) {
   // compute the linear layer

   // convolution input buffer: get access to 1024 samples of each channel: 2048x1 samples
   // convolution outputbuffer: 1x3000
   // compute the convolution: 3000x2048 * 2048x1 + 3000x1
}

template <typename Context>
__device_fct void conv_activation_task(Context context,
              __device_addr ProcessorParameter* processor_param,
              __device_addr ConvActivationParams* task_param,
              __device_addr float* __device_addr* input,
              __device_addr float* __device_addr* output) {
  // compute the activation functions and multiply

  // get input and output buffers
  // apply activation: relu[0..1500) * sigmout[1500..3000)

}
```

LSTM

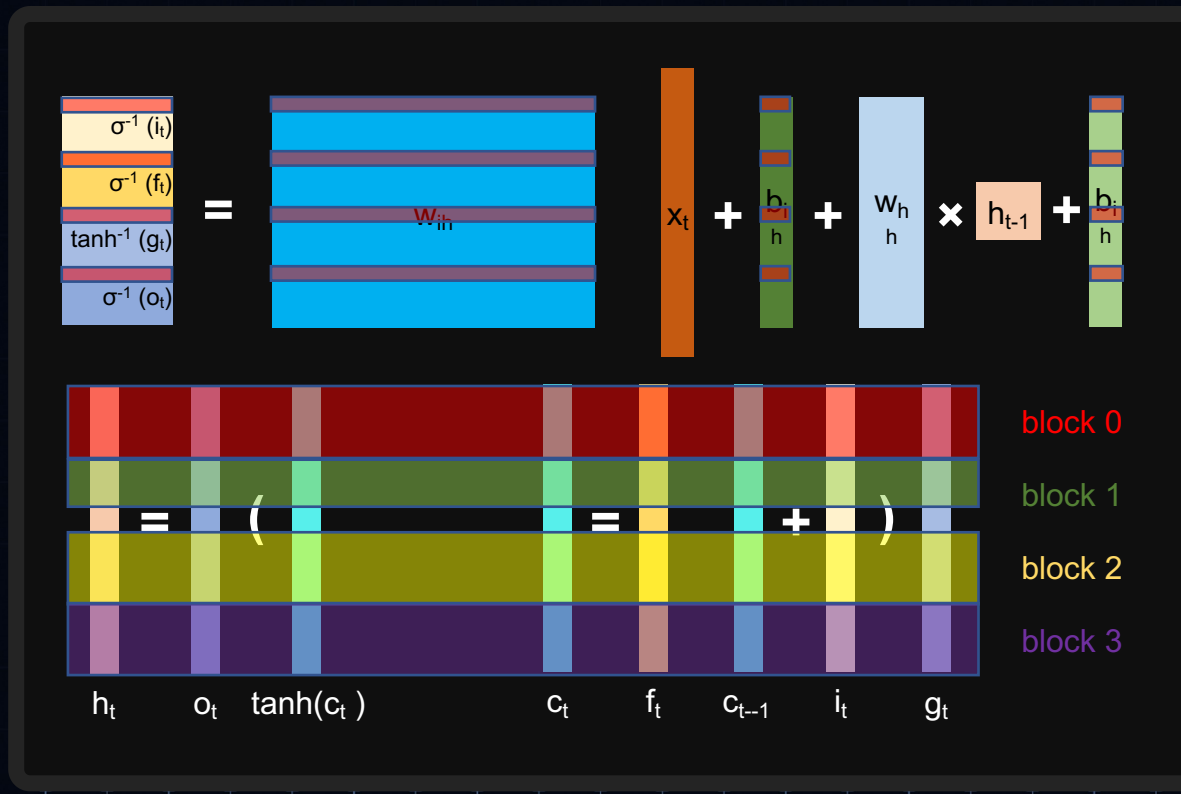
- Provided as a building block by our SDK
- Now also supports the feature of distribution across multiple blocks to support larger weight matrices
- Every layer needs its own task, when spread over multiple blocks (output vector is produced by multiple blocks and they need to synchronize)
- Process function requires knowledge about the number of blocks and block size working on the LSTM step
- The GroupSize determines how many thread cooperatively work on a single output element
- The object uses two hidden state and cell state vectors for double buffering (input and output)
    - The counter variable is used to swap between the two arrays

```
template <uint32_t INPUT_SIZE, uint32_t HIDDEN_SIZE>
struct LSTMLayer {
    Matrix<4 * HIDDEN_SIZE, INPUT_SIZE> m_weight_ih;
    Matrix<4 * HIDDEN_SIZE, HIDDEN_SIZE> m_weight_hh;
    Vector<4 * HIDDEN_SIZE, TYPE> m_bias_ih;
    Vector<4 * HIDDEN_SIZE, TYPE> m_bias_hh;
    Vector<HIDDEN_SIZE, TYPE> m_hidden_state[2];
    Vector<HIDDEN_SIZE, TYPE> m_cell_state[2];

template <uint32_t Layer, uint32_t Blocks,
          uint32_t BlockSize, uint32_t GroupSize,
          typename Context  typename OutputIterator>
void Process(Context& context,
          float* smem,
          OutputIterator output,
          float const* input,
          uint32_t counter);

};
```

LSTM efficiency

- Weight matrices and biases combine i, f, g, o
- Every block is responsible for part of the output
- Every block needs subsections of the weight matrices and biases
- All taken care of by our SDK

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$



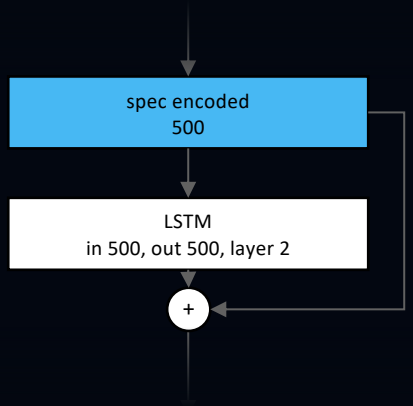
LSTM

- Adding the input to the output would require another task as the computation is carried out by multiple blocks
- Or requires knowledge of the internal workings of the LSTM building block and re-loading the elements computed by the block
- We support an OutputIterator, which can simply add the input to the output when the write operation of the LSTM block is carried out
  - This allows to save a complete tasks
  - Avoids and additional roundtrip through global memory

```
template <uint32_t Layer, uint32_t Blocks,
          uint32_t BlockSize, uint32_t GroupSize,
          typename Context  typename OutputIterator>
void Process(Context& context,
          float* smem,
          OutputIterator output,
          float const* input,
          uint32_t counter);


struct AddOperatorArray {
          float* m_out;
          const float* m_add;
};
```

Have a look

Write the LSTM with ResNet structure



```
template <typename Context>
__device_fct __forceinline_fct void pre_spec_waveform_lstm_task_0(Context context,
       __device_addr ProcessorParameter* processor_param,
       __device_addr PreSpecWaveformLstmParams* task_param,
       __device_addr float* __device_addr* input,
       __device_addr float* __device_addr* output) __device_addr {
    // spectral and waveform LSTM layer 0 500-> 500

    // spectral banch lstm layer 0
    // waveform branch lstm layer 0
}
template <typename Context>
__device_fct __forceinline_fct void pre_spec_waveform_lstm_task_1(Context context,
       __device_addr ProcessorParameter* processor_param,
       __device_addr PreSpecWaveformLstmParams* task_param,
       __device_addr float* __device_addr* input,
       __device_addr float* __device_addr* output) __device_addr {
    // spectral and waveform LSTM layer 1 500-> 500

    // spectral banch lstm layer 1
    // waveform branch lstm layer 1
}
```

RMS Norm

Operates on a single block for efficient reduction in shared memory

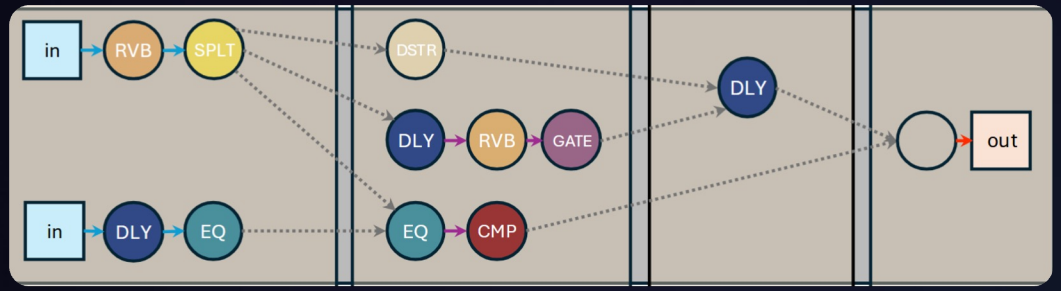
$$y_i = \frac{x_i}{RMS(x)} \cdot \gamma_i$$

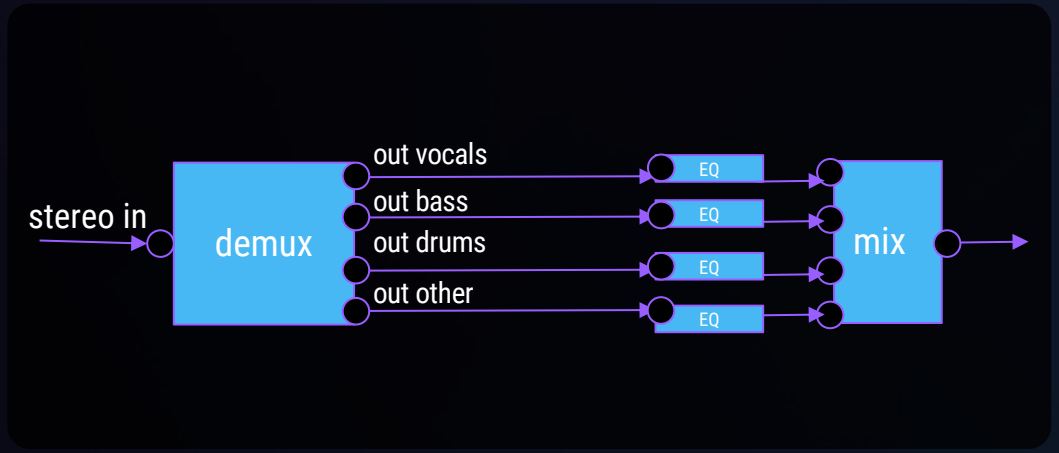
$$RMS(x) = \sqrt{\epsilon + \frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

```
template <uint32_t FEATURE_DIM, typename TYPE>
struct RMSNorm<FEATURE_DIM, TYPE> {
  Vector<FEATURE_DIM, TYPE> m_weights;

  template <uint32_t BLOCK_DIM, typename Context>
  void Process(Context& context,
          float* smem,
          VecType& out,
          VecType const& in, float epsilon = 1e-8);
};
```

GPUA Implementation:
Host Setup

Processing Graph





- GPU Audio works on processing graphs, which need to be directed acyclic graphs (DAG).
- Each node corresponds to a processor, each edge to a buffer going from one processor to another.
- We call the connection points ports.

- The demucs processor has one stereo input and 4 stereo outputs ports.
  - E.g. to apply different equalization settings for each source

Tasks

- Every processor can be run any number of tasks
- Every task must be specified as a separate processor step

```
// final declaration of the processor (in a cu file)
DeclareProcessorStep(DeviceProcessor, 0, my_process_0, float, ProcParam, TaskParam0);
DeclareProcessorStep(DeviceProcessor, 1, my_process_1, float, ProcParam, TaskParam1);
DeclareProcessor(DeviceProcessor, 2);
```

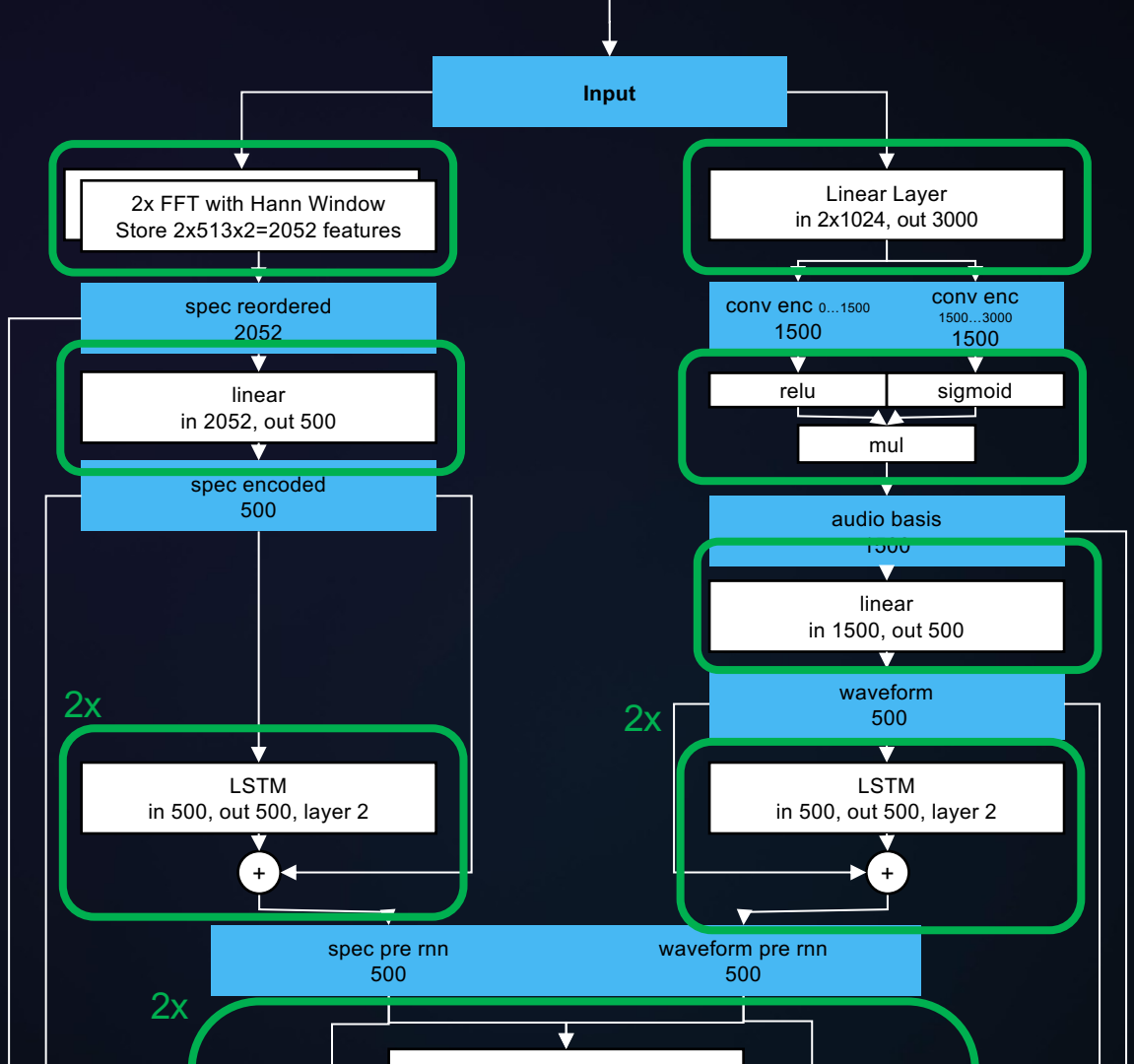
- Tasks can have different number of blocks, threads per block and parameters (TaskParam)

Parameters Example

```
struct ProcessorParameter {
    uint64_t d_hstasnet {};
    uint32_t buffer_capacity {};
    uint32_t ringbuffer_cursor {};
};
struct RMSParams {
    uint64_t p_data_in {};
    uint64_t p_data_norm_out {};
    uint32_t signal_length {};
};

template <uint32_t BLOCKS, uint32_t BLOCK_SIZE, uint32_t GROUP_SIZE>
struct LstmParams {
    static constexpr uint32_t Blocks = BLOCKS;
    static constexpr uint32_t BlockSize = BLOCK_SIZE;
    static constexpr uint32_t GroupSize = GROUP_SIZE;

    uint64_t p_in {};
    uint64_t p_out {};
    uint32_t running_counter {};
    uint32_t signal_length {};
};
```

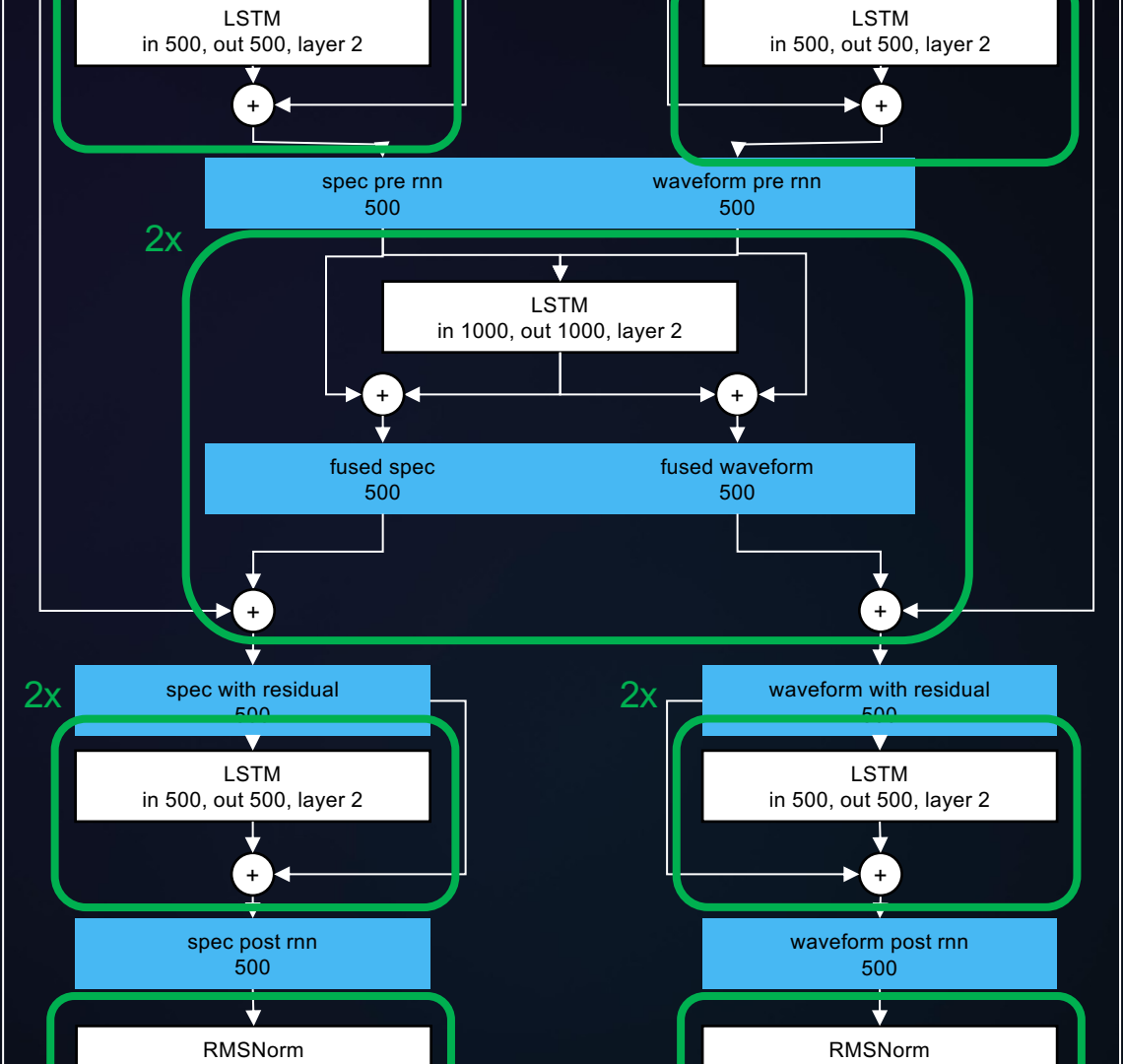
Task setup

Let's have a look at the full source code



Task setup

Let's have a look at the full source code

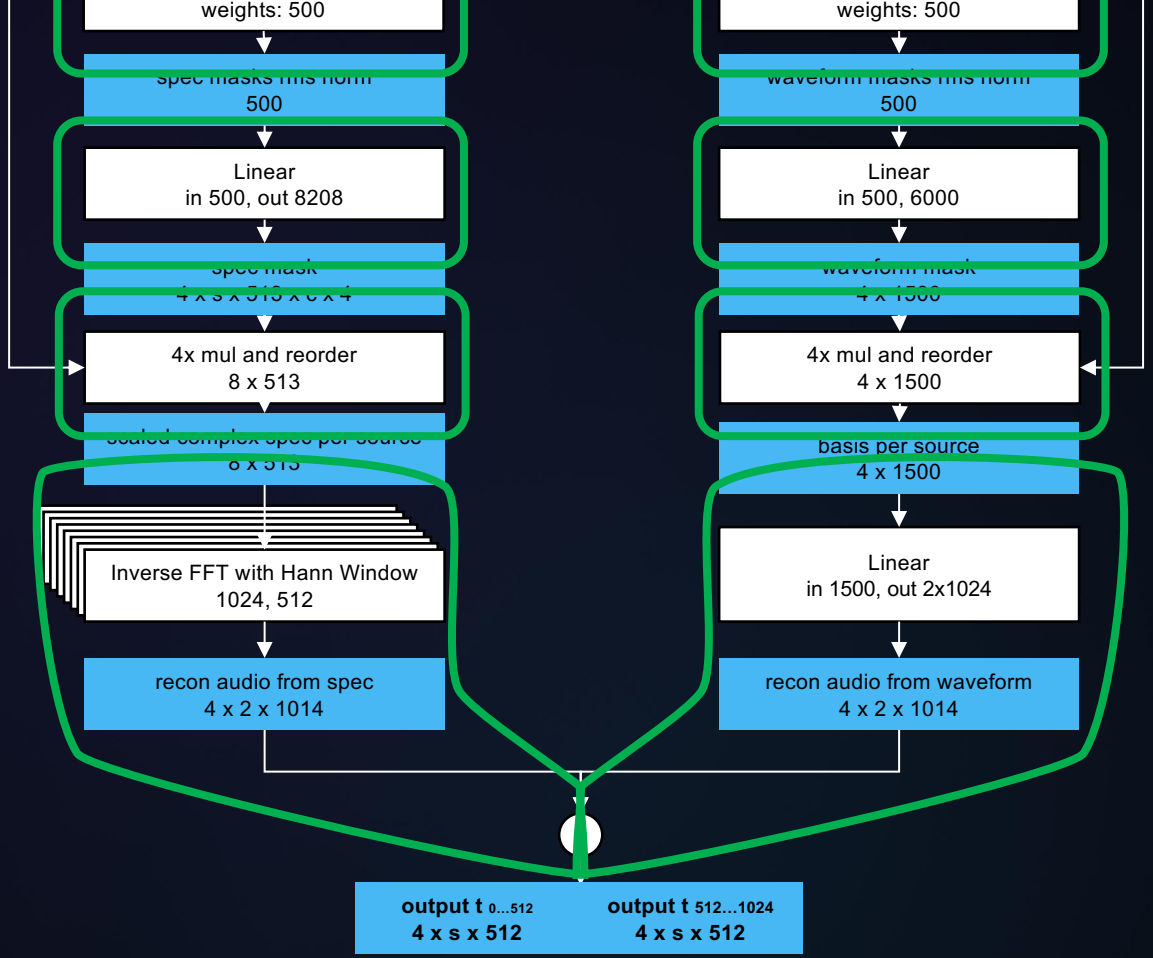


Task setup

Let's have a look at the full source code

Task setup

Let's have a look at the full source code



Tuning

Can vary the number of blocks and threads per task

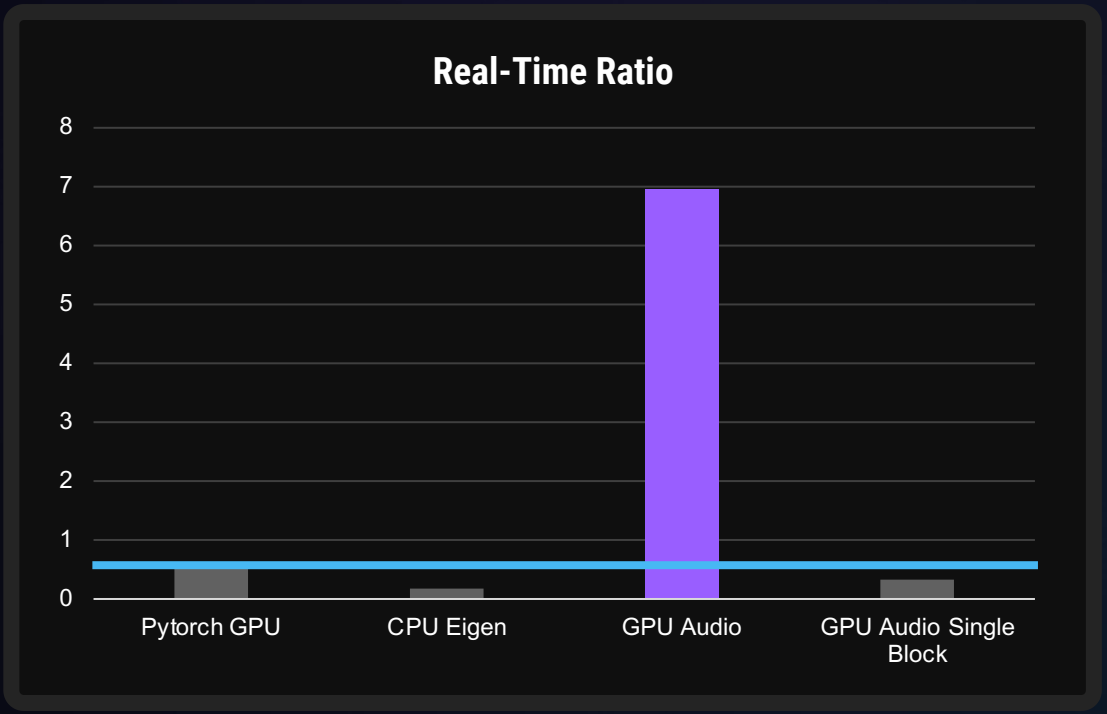
This is usually interpreted as the number of unwanted artifacts a source estimate has with relation to the true source

Could potentially merge tasks together

- Spec branch and waveform branch are very similar
- Tasks have implicit ordering as we do not expose graph-base dependencies through our API
- Our scheduler actually does, we just stuck to a simpler API for now
  With the increase of real-time neural network applications in audio processing we are considering it

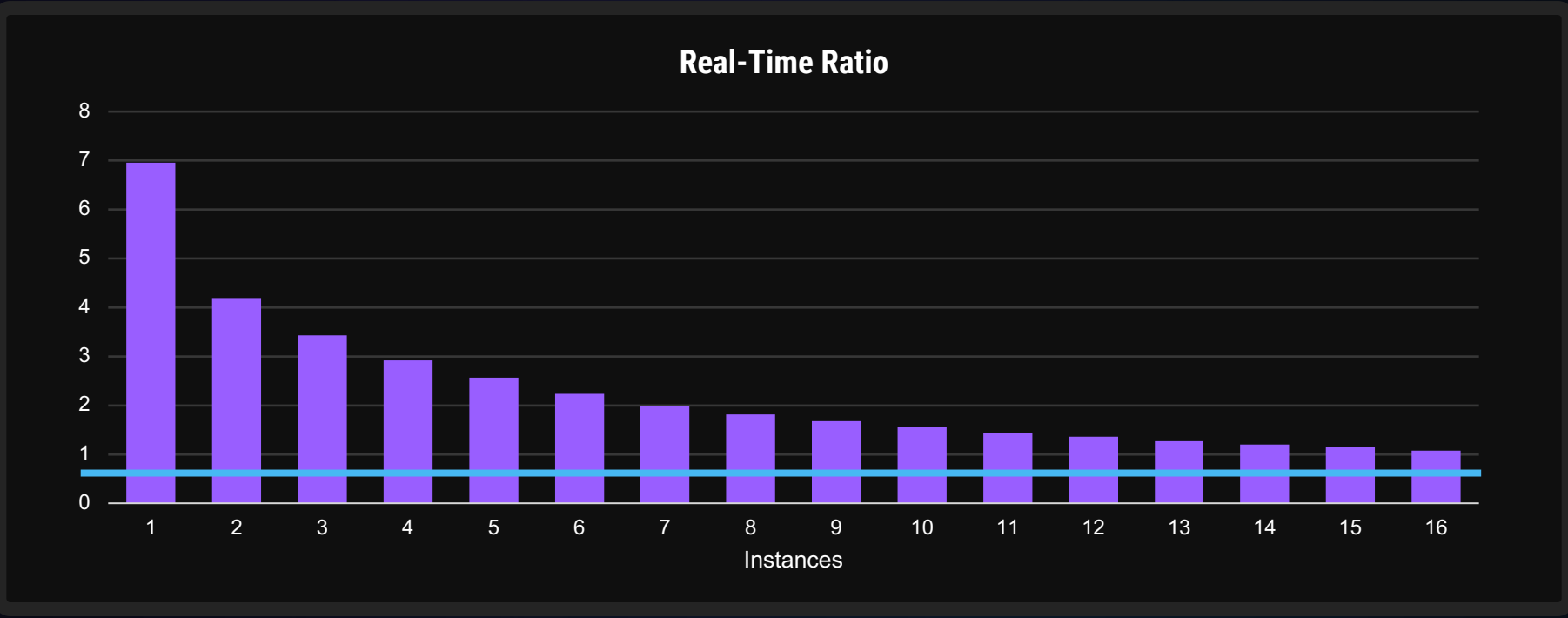
Result Details

NVIDIA 4090 Laptop vs i9-13900HX Laptop



	Real-time Ratio
PyTorch GPU	0.58
CPU Eigen	0.18
GPU Audio	6.96
GPU Audio Single Block	0.33

NVIDIA 4090 Laptop vs i9-13900HX Laptop



Next Steps

Considerations

We took an existing open-source model and brought it to the GPU for real-time execution

- The model latency itself is crucial if you need real-time processing
- If not, we could have used a less causal model but also speed it up with our building block

There may be a space in between

- What if we allowed the model to look ahead 512 or 1024, 2048, or maybe 10000 samples?
- For music playback that may be a reasonable assumption as buffering for a fraction of a second would not disrupt the experience
- The processing cost will increase, but GPUs have a lot of headway if supplied with proper processing setups

Larger networks

- The network sizes of 500x500 are relatively small. What if we used the processing power of the GPU and made them larger?

Get Started with GPU AUDIO Today

Build your own GPU AUDIO-powered HS-TasNet app

Explore real-time source separation and test its performance in your workflow

Acquire a commercial license

from GPU AUDIO to bring your innovation to market — from Pro Audio to Automotive and beyond.

Collaborate with our team for custom integrations

whether you need support for new OS, hardware, or unique use cases.

Optimize your models with our expert assistance

in dataset training, fine-tuning, performance optimization, and latency reduction for your hardware and setup.

GPU.AUDIO

gpu.audio