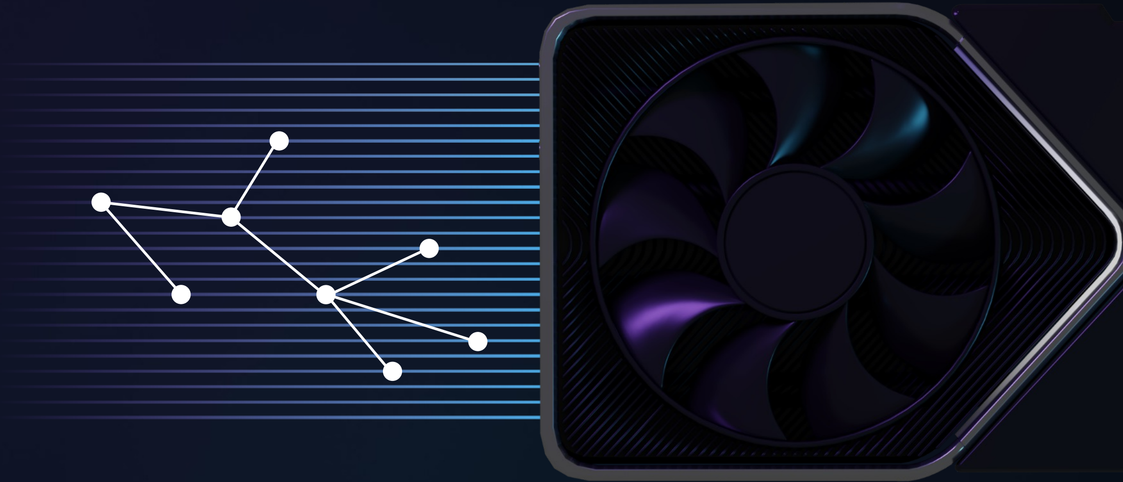
GPU Powered
Neural Audio



Workshop | November 2024

gpu.audio

Neural Amp Modeler

- Open-Source deep learning guitar amp and pedal modeler
- Available as a VST3/AU plugin for Mac/Win as well as a standalone app
- Homepage: https://www.neuralampmodeler.com/
- Author: Steven Atkinson

Guitar Amps Modeling

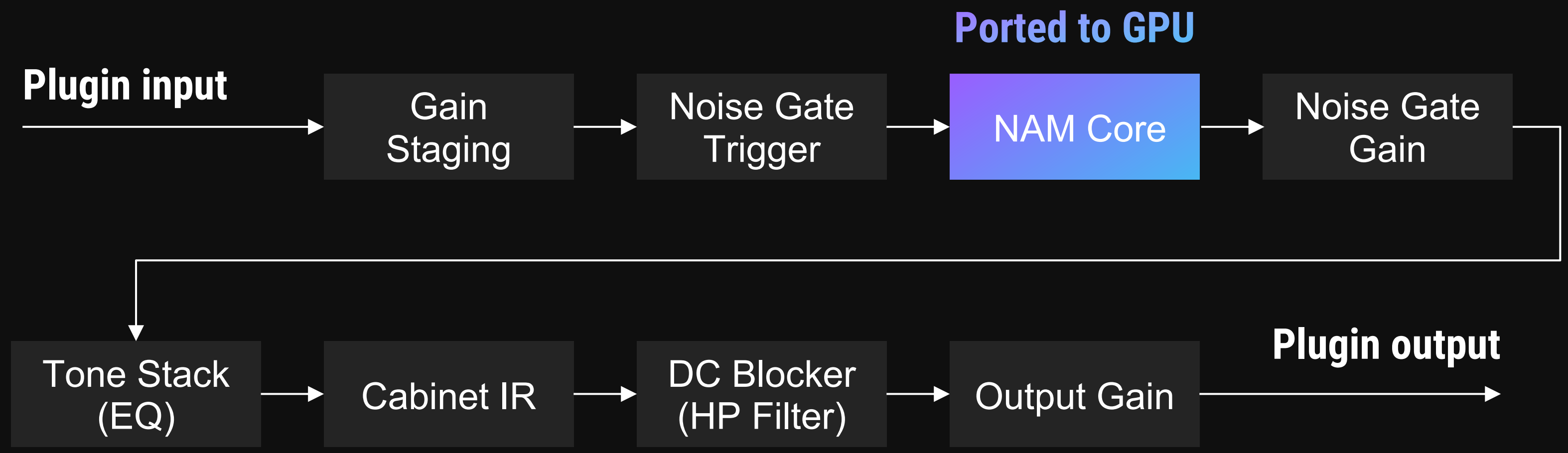
- Guitar amp is a highly non-linear device
- Emulation with conventional modeling methods is complicated due to non-linearities and for each particular amp model should be designed mostly from scratch
- Good fit for the ML approach

Neural Amp Modeler Limitations

Captures amp with the specific settings (knob positions)

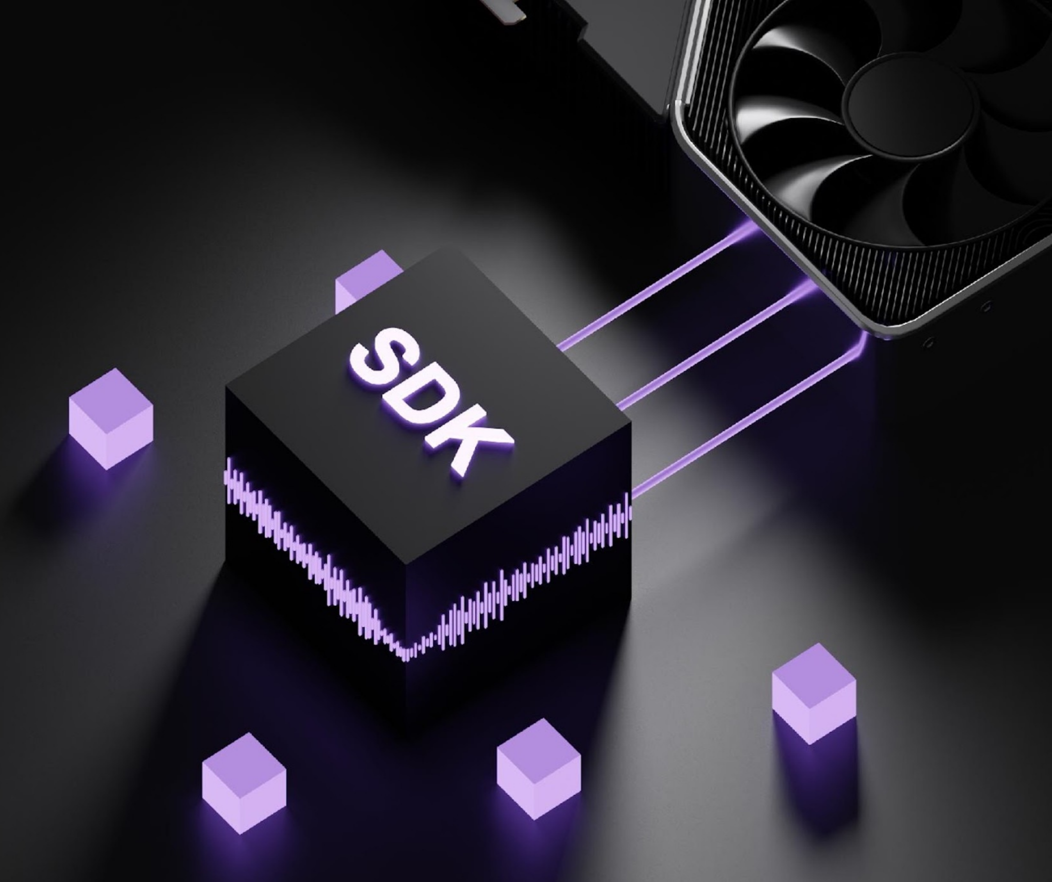
This is why there is no gain knob in the interface

NAM (DSP)


Ported to GPU
Plugin input
Gain Staging
Noise Gate Trigger
NAM Core
Noise Gate Gain
Tone Stack (EQ)
Cabinet IR
DC Blocker (HP Filter)
Output Gain
Plugin output

GPU Audio SDK Overview

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



GPU AUDIO SDK Workflow Schematics

GPU Audio component (audio processing engine)

- Low-latency scheduler
- Implementation of routines provided by APIs
- Proprietary code, provided as a library

Processor API (interfaces for creating audio processors)

- Open header library
- Provides necessary tools for creating your own audio effects
- Uses GPU Audio engine

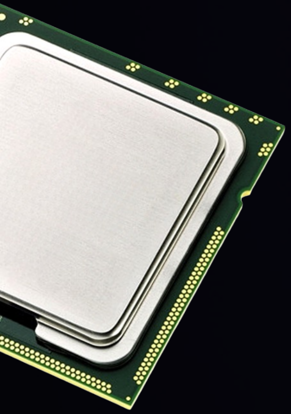
Engine API (interfaces for using audio processors)

- Open header library
- Provides necessary tools for using your own audio effects for processing
- Uses GPU Audio engine

DSP Components Library

- Contains various already implemented filters, partitioned convolution, fft, Neural Network Building Blocks
- Independent of GPU Audio
- Can be used when writing your own audio effects

Cross-platform capabilities



Unified CPU-side interfaces

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

Common device-side C++ style language

- Synchthreads, 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 | ADC24

GPU Audio

- Scheduler
- Memory Management
- Graph Setup and Validation
- Graph Launcher

Graph Launcher

- Instantiation of Processors
- Creation of Processing Graphs
- Data Transfer Control
- Synchronous and Asynchronous Launch

Processing Graph

- The Processing Graph holds multiple processors and their connections (ports)
- Determines an ideal way of scheduling the graph on the respective hardware, optimizing for number of GPU launches, temporary memory requirements, and latency

Processor

- Core processing functionality of a processor, split into task, blocks, and threads running on the GPU
- Parameter passing control
- Memory management and transfer as needed
- Hints for the gpu audio engine about processing characteristics

*internally it's a special case of DAG

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

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

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

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

PrepareForProcess / PrepareChunk — Preparation function for reacting to new input data and providing parameters for GPU execution

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

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

CpuMemoryPointerAllocatePinnedCpuMemory(.
..);

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

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

MemoryManager — 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).

NAM Models

Three different implementations

1 Convnet

- Simple MLP with multiple layers working on current and previous inpu

2 LSTM

- Long short-term memory implementations
- Two hidden layers

3 Wavenet

- Latest version of the model using dilated convolution to combine previous input and data with current inputs
- 2x 10 dilated convolution layers

Wavenet

Van Den Oord, Aaron, et al. "Wavenet: **A generative model for raw audio."** arXiv preprint arXiv:1609.03499 12 (2016)

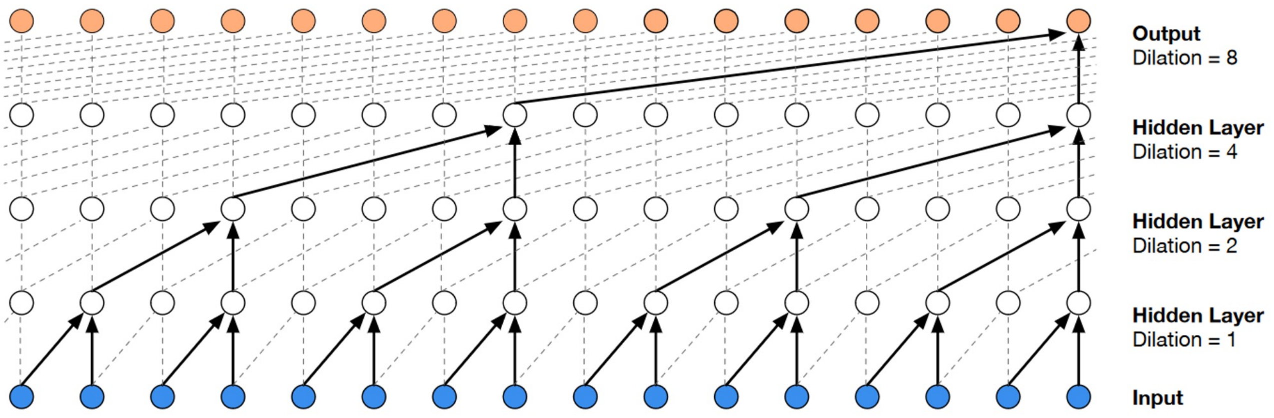
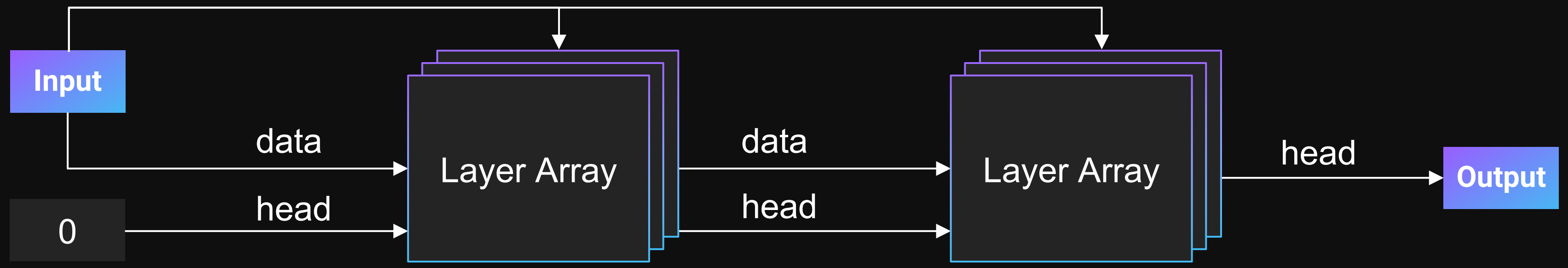
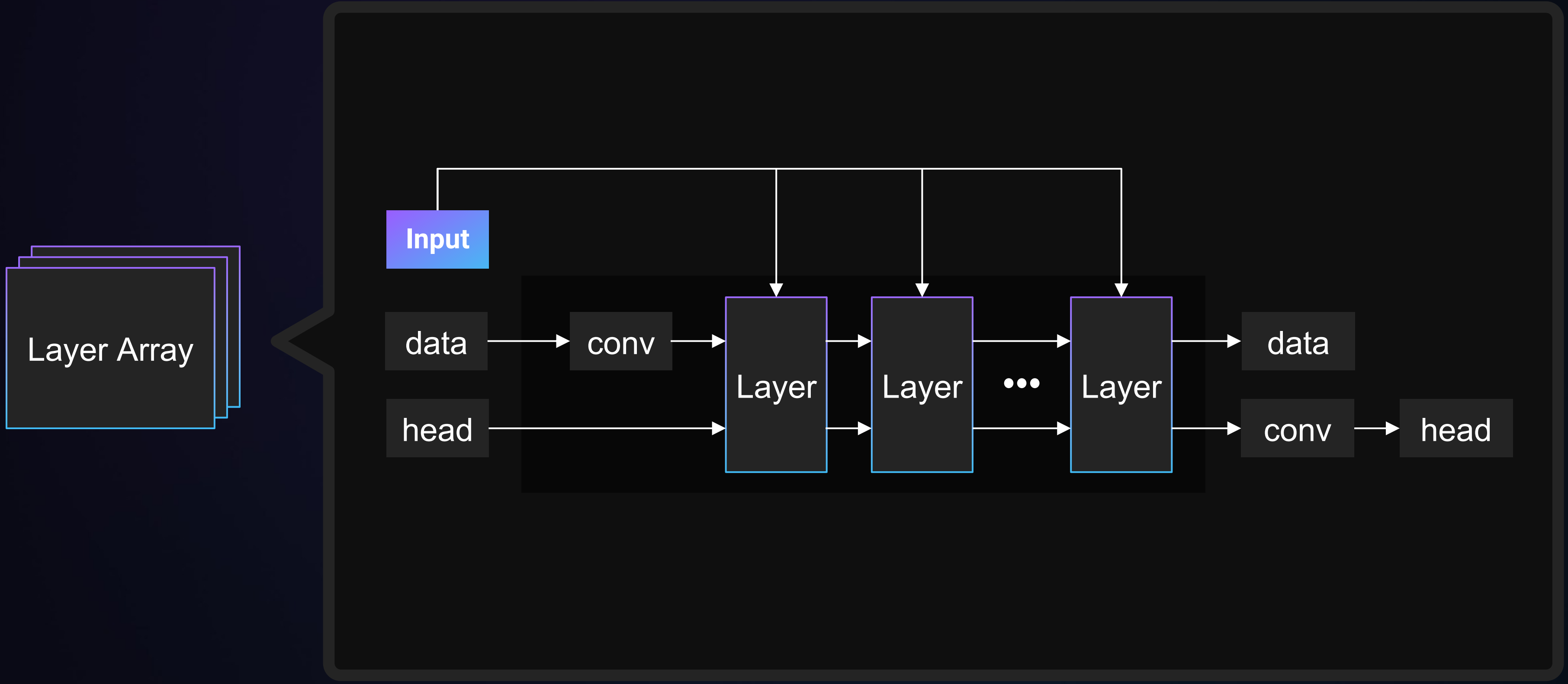


Figure 3: Visualization of a stack of *dilated* causal convolutional layers.

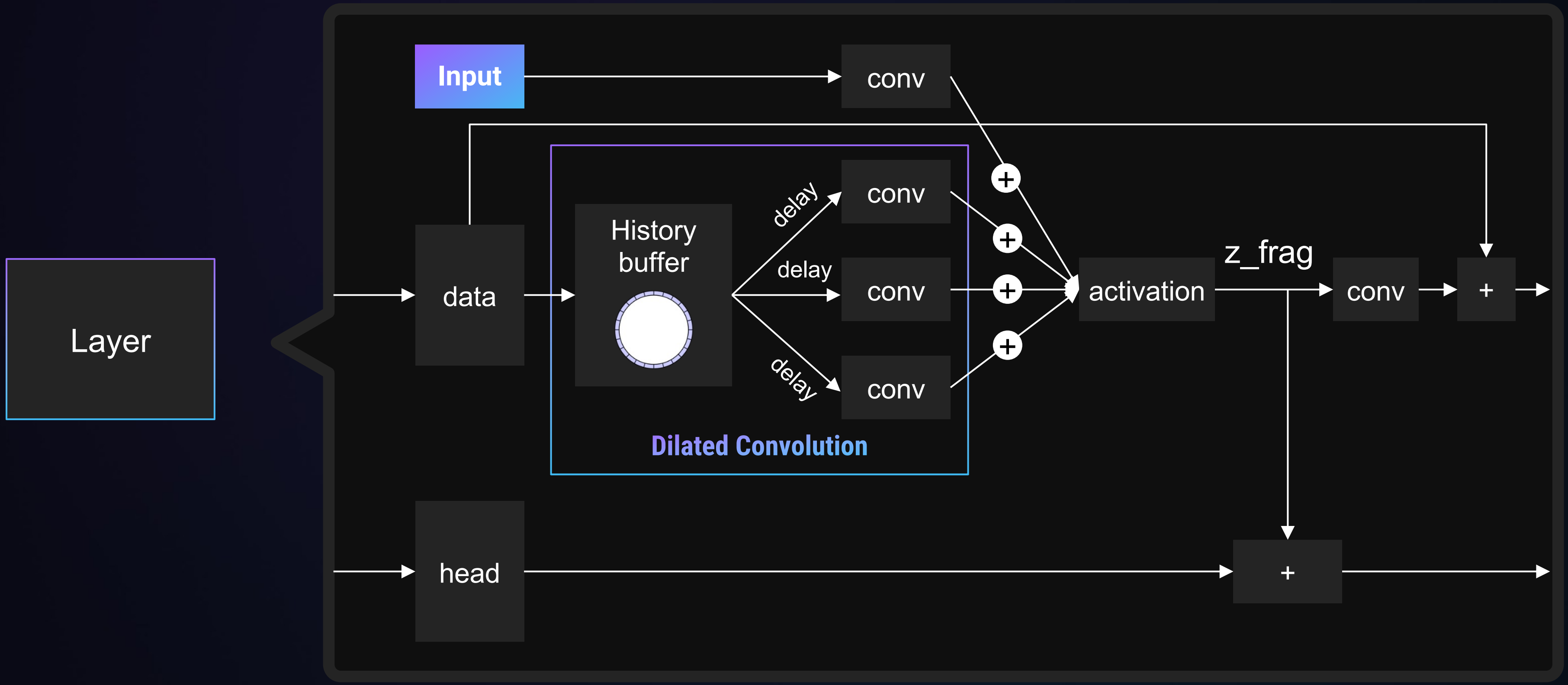
Process: Top level



Process: Layer Array



Process: Layer



GPU building blocks used today

Multichannel Delay Line implemented as a ringbuffer

- Allowing to store arbitrary numbers of channels in a history buffer and access any data

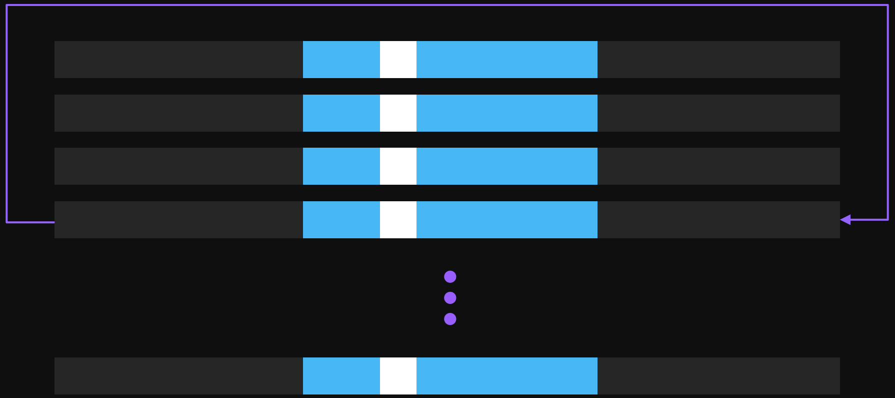
Conv1x1

- Implemented as matrix multiplication with and without bias

Matrix Multiplication

- To implement dilated convolution

Multichannel Delay Line



- Implemented as a ringbuffer
- Size should be chosen such that sufficient history can loaded
- Size must be power of two to allow unit wrap around
- Cursor to capture current position
- Load and Store as vector or matrix

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

  template <class Fragment, class
Context>
  Fragment LoadAsMatrixFragment(Context&
context, uint32_t cursor) const;

  Vector<Channels, Type> Load(uint32_t
cursor) const;

  template <class Context, class
Fragment>
  void Store(Context& context, uint32_t
cursor, const Fragment& fragment);

  void Store(uint32_t cursor, const
Vector<Channels, Type>& data) ;
};
```

Matrix

Matrix multiplication the core of most neural networks' operations

Matrix multiplication typically computed by multiple threads together

- Most often a warp - SIMD group of threads
- Can also be an entire block of threads

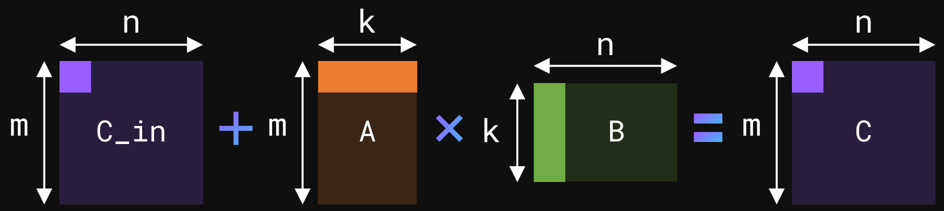
Matrix multiplication mostly limited by memory access nowadays

- Our building blocks help with memory transitions

Matrices can be held in

- shared memory (efficient on-chip memory, accessible by all threads)
- or in registers = fragments (distributed across multiple threads)

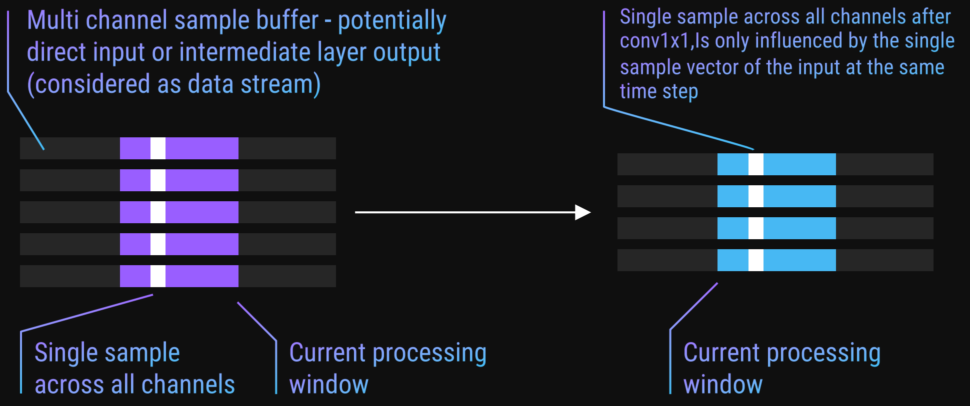
Matrix Multiplication



- Matrix A held in shared memory
- Matrix B held as a fragment or shared memory
- Matrix C held as a fragment
- Matrix C used as accumulator to add on top

```
template <class Context, uint32_t M,
uint32_t N, uint32_t K, typename
TYPE_INPUT, typename TYPE_ACCUMULATOR>
MatrixFragment Multiply(Context& context,
MatrixShared const& matA, MatrixFragment
const& matB, MatrixFragment const&
accumulator = {});

template <class Context, uint32_t M,
uint32_t N, uint32_t K, typename
TYPE_INPUT, typename TYPE_ACCUMULATOR>
MatrixFragment Multiply(Context& context,
MatrixShared const& matA, MatrixShared
const& matB, MatrixFragment const&
accumulator = {});
```

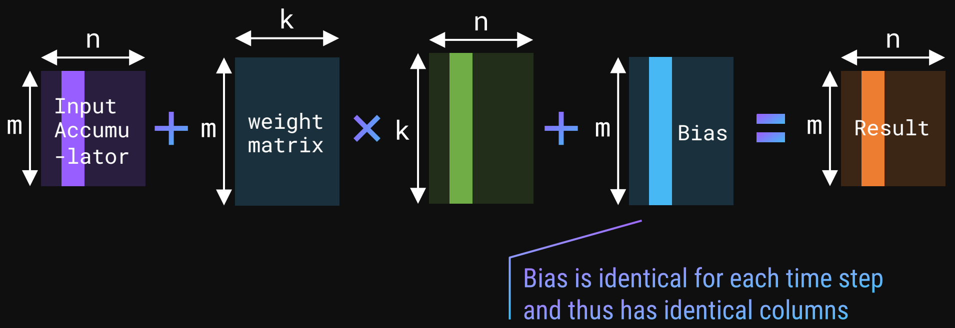
Conv1x1



- Implemented as a matrix multiplication
- Bias given by a vector
- M.. number of output channels
- K.. number of input channels
- N.. number of samples
- Temporary shared memory provided by the class to load the matrix for multiplication
- Input data either in shared memory or as fragments

```
template <uint32_t M, uint32_t N,
uint32_t K, typename TYPE, bool BIAS =
false>
struct Conv1x1 {

void Set(Type const* weights, Type const*
bias);

 template <class Context>
 AccumulatorFragment Process(Context&
context, MatrixBShared const& matB, Smem&
temp, AccumulatorFragment const&
accumulator = {}) const;

 template <class Context>
 AccumulatorFragment Process(Context&
context, MatrixBFragment const& matB,
Smem& temp, AccumulatorFragment const&
accumulator = {}) const;


};
```

Conv1x1



- Implemented as a matrix multiplication
- Bias given by a vector
- M.. number of output channels
- K.. number of input channels
- N.. number of samples

- Temporary shared memory provided by the class to load the matrix for multiplication
- Input data either in shared memory or as fragments

```
template <uint32_t M, uint32_t N,
uint32_t K, typename TYPE, bool BIAS =
false>
struct Conv1x1 {

void Set(Type const* weights, Type const*
bias);

 template <class Context>
 AccumulatorFragment Process(Context&
context, MatrixBShared const& matB, Smem&
temp, AccumulatorFragment const&
accumulator = {}) const;

 template <class Context>
 AccumulatorFragment Process(Context&
context, MatrixBFragment const& matB,
Smem& temp, AccumulatorFragment const&
accumulator = {}) const;


};
```

Device Execution Quick Info

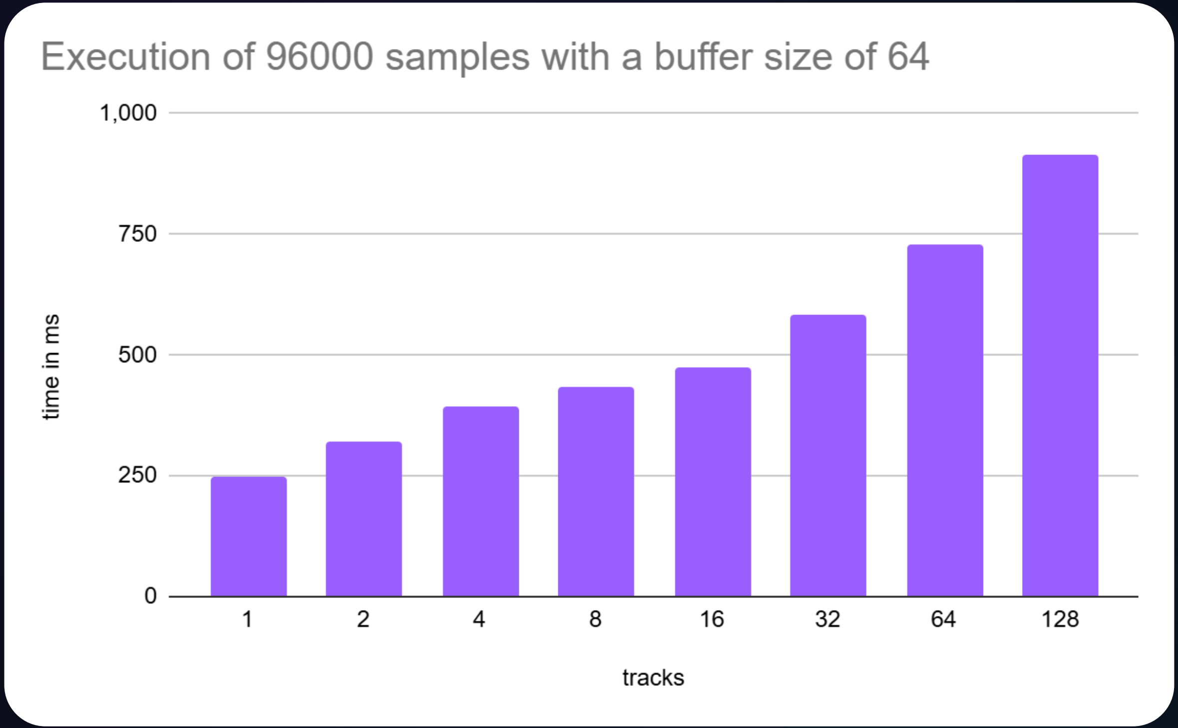
```
class DeviceProcessor {
public:
  template <class Context>
__device_fct void init(Context context,
unsigned int bufferLength) __device_addr;
  template <class Context>
__device_fct void my_process(Context
context, __device_addr ProcParam* params,
__device_addr TaskParam* task_params,
__device_addr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

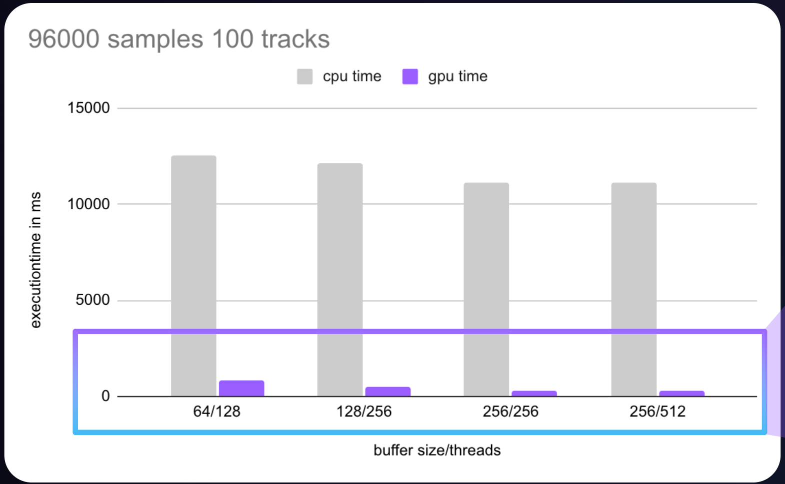
```
// final declaration of the processor (in a cu file)
DeclareProcessorStep(DeviceProcessor, 0, my_process,
float, ProcParam, TaskParam);
DeclareProcessor(DeviceProcessor, 1);
```

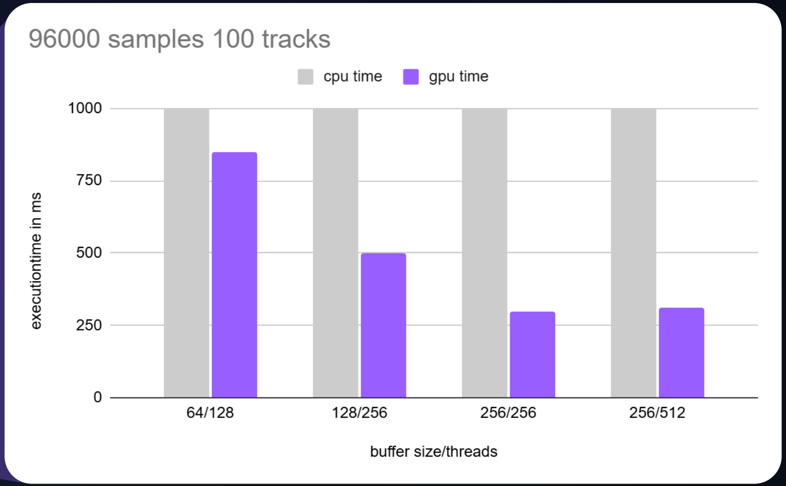
- Each process method needs to be declared (and numbered). The input data type (float) and the parameter types need to be specified.
- The final processor declaration only need to class name and the number of process functions (1 in this case)

Performance info NVIDIA 4090s

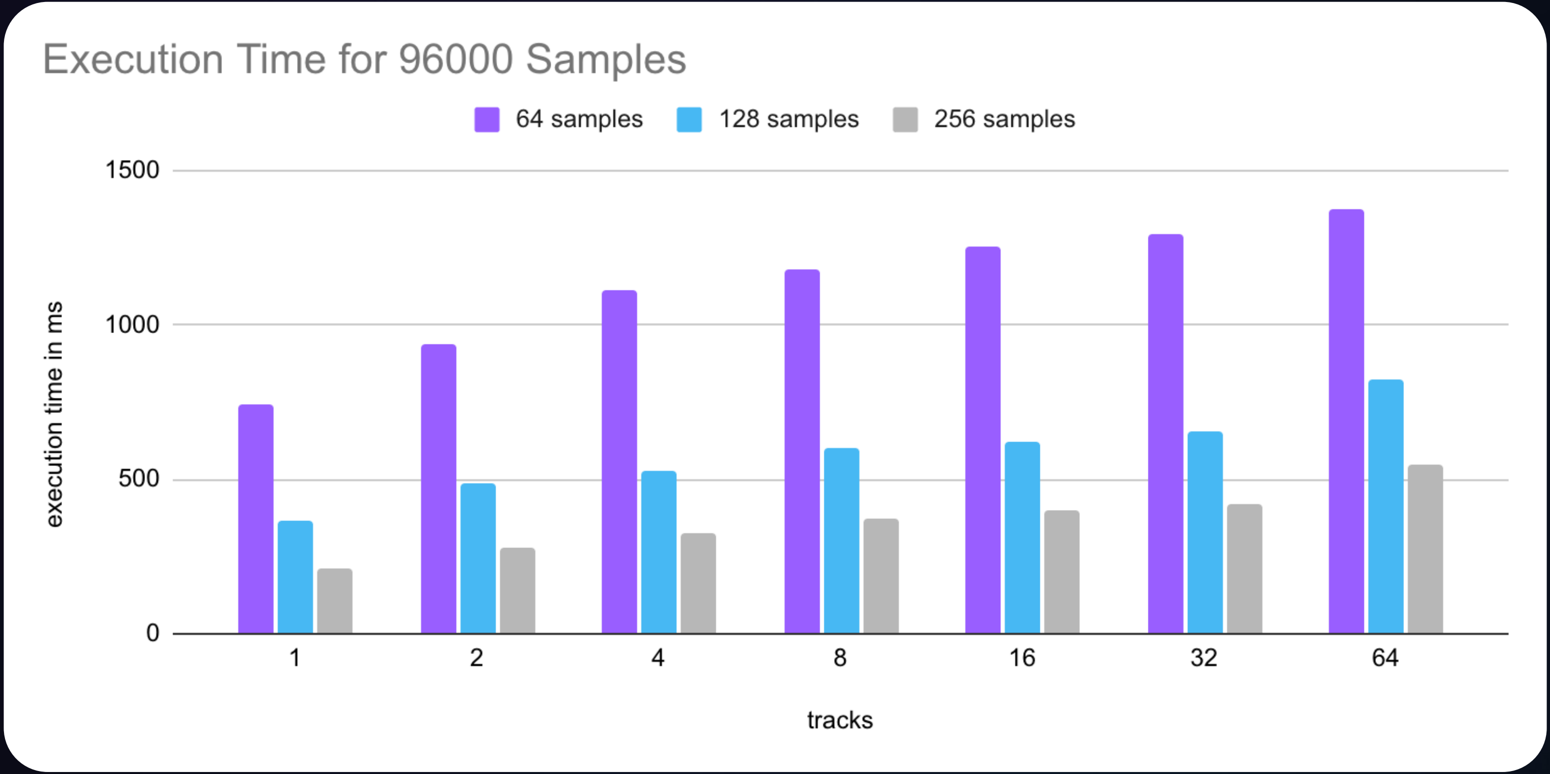


Performance info NVIDIA 4090s

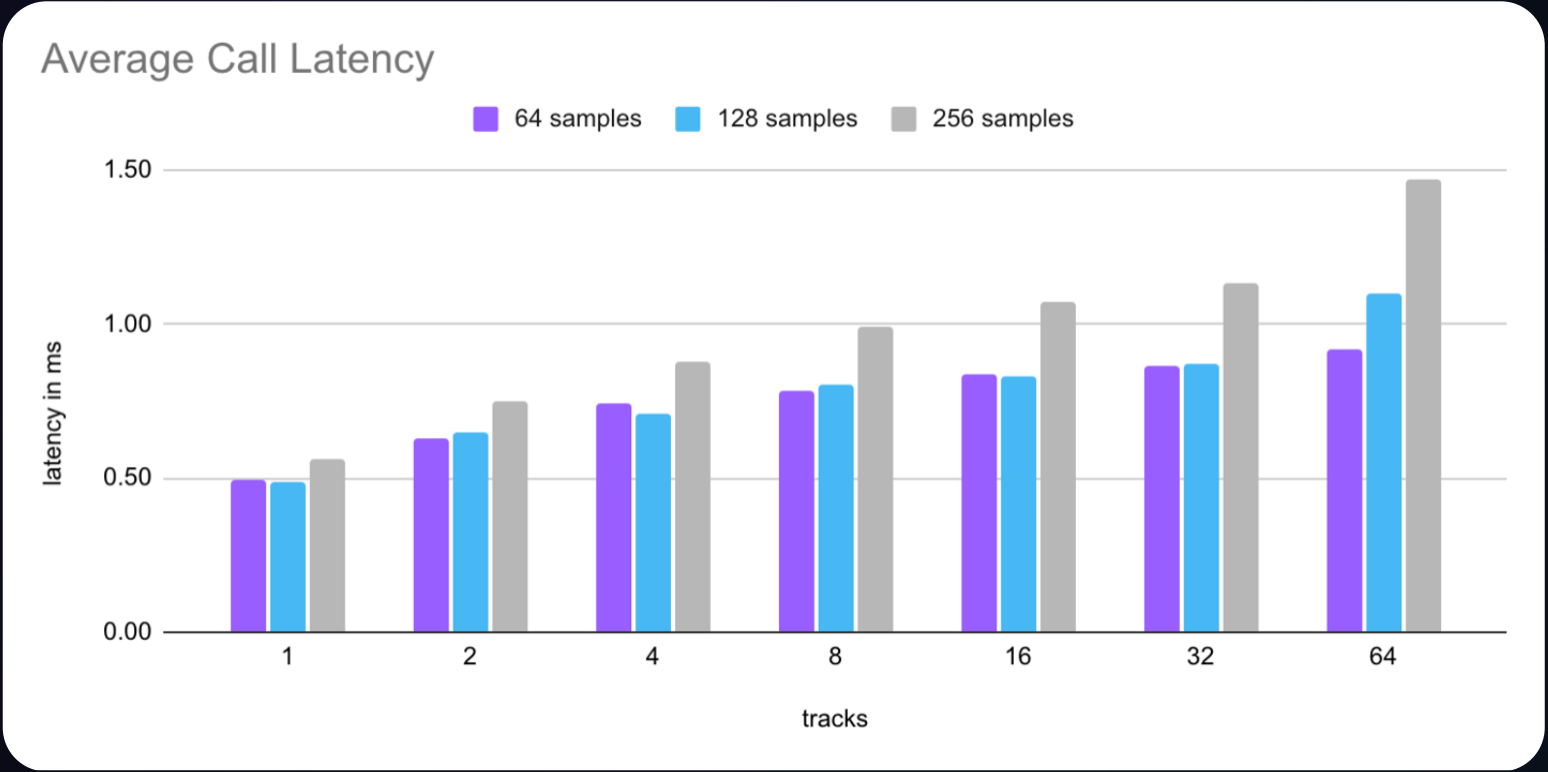




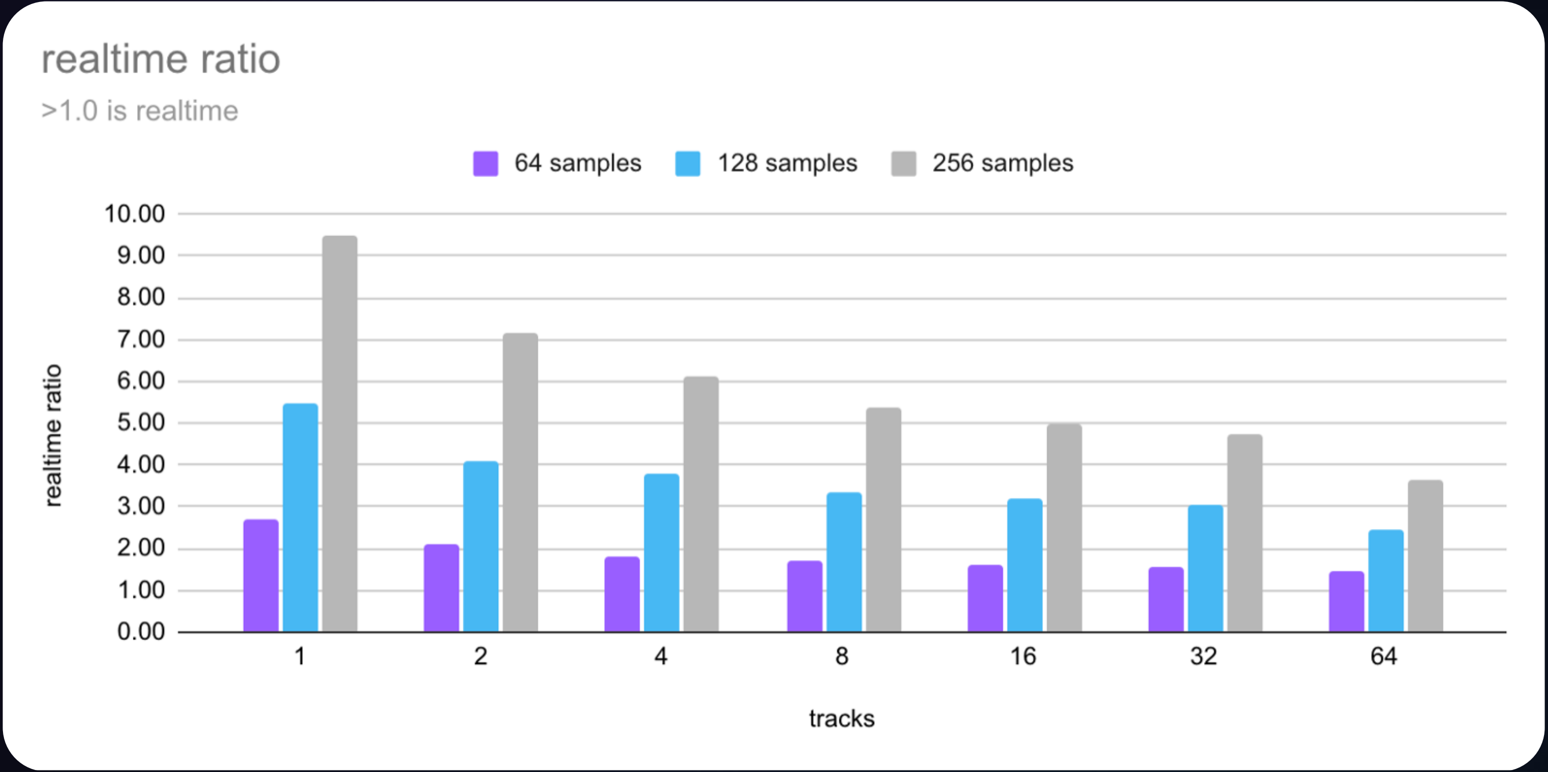
Performance info Mac M2 Max



Performance info Mac M2 Max



Performance info Mac M2 Max



Hands-on demo in Jupyter environment

Sign up credentials:

adc2024.gpu.audio

Username: adc2024
Password: 8uWpaR36zwUXWDBcg4eeZGK5

Future and challenges to solve

- Target architecture is a Cartesian product of CPU_arch x GPU_arch x OS
- Different versions of compilers, CUDA, HIP, metal, etc
- Current amount of target profiles that we are using internally is ~300
- Implementation of toolchain and profile on-demand generation outside of GPU Audio internal infrastructure

Result: public GPU Audio SDK Preview Release

Q&A

gpu.audio