Zurich University
of Applied Sciences

School of
Engineering
InES Institute of
Embedded Systems

GPU Based Audio Processing Platform with AI Audio Effects.

Zurich University of Applied Sciences

Institute of Embedded Systems

Authors	Simon Schneider
Version	1.0
Last changes	October 8, 2024

Contact Information
c/o Inst. of Embedded Systems (InES)
University of Applied Sciences Zurich
Technikumstrasse 22
CH-8401 Winterthur

Tel.: +41 (0)58 934 75 25
Fax.: +41 (0)58 935 75 25

E-Mail: scso@zhaw.ch
Homepage: http://www.ines.zhaw.ch

Zurich University
of Applied Sciences

School of
Engineering

DECLARATION OF ORIGINALITY

Master's Thesis at the School of Engineering

By submitting this Master's thesis, the undersigned student confirms that this thesis is his/her own work and was written without the help of a third party.

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the Master's thesis have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

City, Date: Signature:

....

The original signed and dated document (no copies) must be included after the title sheet in all ZHAW versions of the Master's thesis submitted.

Zurich University
of Applied Sciences

Abstract

Parallelising real-time audio effects requires complex task management and synchronisation. GPUs are optimised for parallel processing while still retaining flexible scheduling comparable to a CPU As a result, it combines the strengths of both DSPs and CPUs in a single device. In addition, recent trends advocate using AI audio processing algorithms, which work best on GPU architectures.

This thesis presents an implementation of an embedded GPU-based audio processing framework on an Nvidia Jetson hardware platform. It can combine neural network inference and other audio effects into signal graphs that process within periods as small as 32 frames (0.667ms).
The signal graph does not restrict the number and combination of parallel and serial audio effects as long as the real-time limit is met. Therefore, the framework has been tested on large numbers of parallel channels, as found in a mixing console, and complex routing options available in high-end audio effect processors, such as the Neural DSP Quad Cortex.

Launching GPU work using the CUDA graph API produces better stability and performance than was observed using the CUDA stream API in a 2017 study.[1] Processing a signal graph that fully utilises the Jetson's resources by mimicking a 64-channel mixing console on a 128-frame (2.67ms) period has a higher than 99% success rate. However, occasional stalling on the GPU can produce worst-case execution times of up to 20ms, regardless of the loaded audio effects. As a result, the framework can not yet be classified as real-time capable.
Further study of the CUDA scheduler and improvements to the operating system and audio driver may be able to achieve real-time capability in the future.

Contents

1. Introduction

CPUs dominate the consumer-grade audio processing market, even though they are not necessarily the best choice for digital audio processing. Thanks to years of scheduling optimisation and high clock speeds, they efficiently handle the complex control flows of signal chains built from audio plugins within a Digital Audio Workstation (DAW). However, they are not well-suited for Single Instruction, Multiple Data (SIMD) operations, which are essential for audio processing.[2] As a result, live sound engineering devices still heavily rely on dedicated Digital Signal Processors (DSPs) for their high throughput and reliability.

GPUs combine the high throughput of DSPs with the flexibility of CPUs, making them an appealing option for audio processing tasks. However, there is a drawback.
Initially designed for digital image processing, GPUs are optimised to handle embarrassingly parallel tasks in large volumes. In contrast, audio processing tasks often involve serial dependencies and, in real-time processing, operate on very small data sizes.

Despite these challenges, the alternatives either lack the flexibility to adapt to modern software development trends or are not optimised for the task at hand. With the CPU's single-core speed advances stagnating, it may be time to consider changing course.

1.1. Initial Situation

In 2023 the company GPU Audio has demonstrated that offloading suitable audio effects to a GPU is feasible.
Using their proprietary Application Programming Interface (API), the Vienna Power House extension to the Vienna MIR Pro 3D plugin by VSL can process up to 1760 positional Impulse Response (IR) in real-time on a single GPU. The plugin processes with a minimum buffer size of 128 samples, resulting in a latency of 2.67ms at a sample rate of 48kHz, in addition to the audio interface's round-trip latency. While impressive, the process latency is already at the upper limit of what would be acceptable for a device employed for live sound engineering, as described in Section 2.3.

In their presentation at the audio developers conference in 2023, GPU Audio mentioned that they are working on getting their API into the hands of dedicated hardware manufacturers. However, as of the time of writing, I have not found any mention of GPUs being used for live sound engineering.

1.2. Motivation

This thesis is driven by two main factors:

Firstly, there is an academic motivation to evaluate the potential of GPUs as the next platform for audio processing, following the dominance of DSPs and CPUs in the market.
A 2021 study on the evolution of CPUs concludes that Dennard scaling has not been maintained since the mid-2000s.[3] Due to physical hardware limitations, the annual increase in clock speed has plateaued, leading to the development of multi-core CPUs. As a result, modern audio processing hardware depends on multi-core CPUs or multiple DSPs to achieve higher throughput.
However, audio effects in single-channel signal chains are usually arranged in series. Therefore, parallelism has to be achieved using SIMD operations and and parallel channel processing.
With the introduction of the graph API in CUDA-10.0, the GPU appears to be a perfect candidate for this challenge. It promises to reduce the overhead of launching repetitive tasks at the cost of a longer setup time required to instantiate the graph.

Additionally, the market for digital audio effect processors with AI inference capabilities has grown significantly in recent years. Since all these neural networks are trained on GPUs, running the inference on the same hardware would be logical.

Secondly, I have a personal desire for a general-purpose audio processing platform that is customizable, scalable, and can process audio in real-time.
Currently, my band depends on a complex setup of devices for live performances:

- one Neural DSP Quad Cortex guitar amplification
- one Darkglass Alpha Omega Ultra for bass amplification
- one Behringer X32 mixing console with an SD8 stage box for signal routing
- six PM 16 personal monitor mixers for individual InEar mixes
- one laptop for playing backing tracks and controlling devices via MIDI

The coordination of all these devices is time-consuming.
In the GPU, I see the potential to power an audio processing platform with the flexibility to handle the tasks of all the listed devices and the scalability to avoid the need for multiple devices.

1.3. Goal

This thesis aims to create a proof of concept for a GPU-based audio processing platform capable of real-time audio processing.
Using an NVIDIA® Jetson Orin™ NX in combination with an off-the-shelf audio interface, the platform is evaluated for its ability to:

- handle multi-channel audio processing similar to a DSP-based mixing console
- efficiently schedule complex signal graphs like a CPU
- run real-time neural network inference as an audio effect like the Neural DSP Quad Cortex

1.4. Target Audience

This document is intended for readers with a basic understanding of computer science and digital audio processing. Prior knowledge of GPU programming is optional as all relevant concepts are explained in the next chapter.

2. Fundamentals

This chapter provides background information on the hardware and software relevant to this thesis.

2.1. Hardware

Consumer-grade computers are powerful enough to process audio in real-time, but they are not optimised for low latency or reliability. As a result, nearly all professional real-time audio processing systems utilise dedicated hardware. The upcoming sections will offer an overview of the hardware used in this thesis.

2.1.1. GPU Architecture

All GPU-related information in this section is based on the NVIDIA Ampere architecture. [4]

Nvidia GPUs consist, at the lowest level, of so-called CUDA cores. However, when compared to CPU architecture, a CUDA core is more akin to a functional unit.[2] Therefore, the number of CUDA cores on a GPU relates to the total number of parallel arithmetic operations it can hypothetically perform.
The cores are grouped into a hierarchical structure of fixed-sized segments to enable efficient work scheduling and allow cross-compatibility between different GPUs.
From the smallest to the largest:

- 32 cores are grouped into an Streaming Processor (SP)
- 4 SPs are grouped into an Streaming Multiprocessor (SM) (see Figure 2.1a)
- 2 SMs are grouped into a Texture Processing Cluster (TPC) (see Figure 2.1b)

Large GPUs further combine TCPs into Graphics Processing Clusters (GPCs), but the layout varies between devices.

(a) SM Architecture [4] (b) TPC Architecture [4]

Figure 2.1.: Ampere Architecture [4]

Each SP has an independent scheduler to execute one SIMD operation on 32 data elements per clock cycle. In CUDA, each generated instruction is run on a single thread. The resulting group of 32 threads is called a Warp.

To fully utilise the SP's computing power, the scheduler must always have data for the next warp ready as soon as the previous has finished. If the scheduler takes longer to fetch data needed for kernel execution than to compute the result, the code has a memory access bottleneck as shown in Figure 2.2. Conversely, the code has a compute bottleneck if the kernel takes longer to compute the result. While balancing the memory access and compute time is important, it may not always be possible. For this reason, developers should give the scheduler multiple warps to work with. That way, it can pipeline memory access and computation to hide excessive latencies on both sides. The downside of which is, that the warps can be executed out of order.

For an embarrassingly parallel algorithms like point-wise array multiplication, this is not a problem. Each thread receives an element at the same index from both arrays to process and writes the result back in-place or to a new array. As a result, each thread can work completely independently.
For algorithms that require inter-thread dependencies, the SM has a shared memory (L1 cache) that can be used to communicate between threads running on all the 4 SPs within the SM.
The grouping of SMs into TPCs and then into GPCs offers further advantages for large-scale graphics rendering. Since this thesis concerns itself with neither large scale computation nor graphics rendering, these topics will not be discussed further.

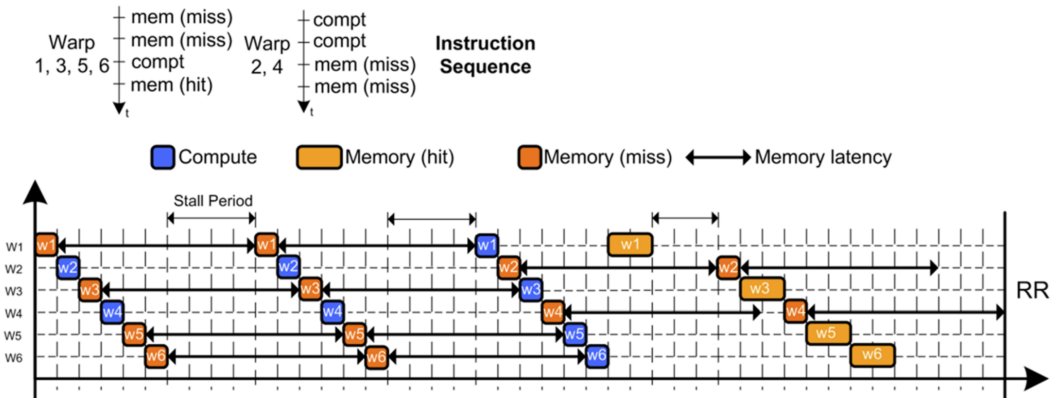


Figure 2.2.: Visualisation of a round-robin warp scheduler with a memory bottlenck.[5] Each warp W1 to W6 comprises four instructions as defined at the top of the image. The horizontal axis represents the execution time of the warps. Both compute and memory instructions with cache hit have a very short execution time. Memory instructions with cache miss have a longer execution time and are prone to introduce pipeline stalling. This example shows how long memory access times, combined with too few available warps, result in pipeline stalling. Other scheduling strategies presented in the paper can reduce the number and duration of the pipeline stalls.

Jetson Orin NX

The NVIDIA® Jetson Orin™ NX (subsequently referred to as Jetson) brings AI supercomputer performance to the edge in a compact system-on-module (SOM) which is smaller than a credit card. [6]
One of the main concerns about using GPUs for audio processing that I noticed being voiced in forums and conferences is the latency introduced by the PCIe bus when transferring real-time audio data between the GPU and the CPU. On the Jetson, this is not a problem, as the CPU and GPU reside on the same chip and have direct access to the same shared memory.

For a project like this, one would usually work with a Jetson developer kit, which includes a carrier board with all the necessary peripherals.
However, since the ZHAW Institute of Embedded Systems, where I am working on this project, has recently redesigned their custom modular vision system carrier board Modular Vision System carrier board, I will be using it instead.

The Jetson has an 8-core ARM Cortex-A78AE CPU and a 1024-core NVIDIA Ampere GPU.[6] The cores are grouped according to the Ampere architecture into 32 SPs, 8 SMs and 4 TPCs. Both the GPU and the Memory clock rate can reach up to 918MHz.
According to the output of the CUDA deviceQuery, the Jetson can handle the expected 1024 threads per block but only 1536 threads per SM. This is something to remember when choosing CUDA kernel launch arguments further on.

2.1.2. Audio Interface

The audio interface is the hardware that contains the Analogue-to-Digital Converters (ADCs) and Digital-to-Analogue Converters (DACs) that capture and playback audio signals. It can either be an external device connected through a peripheral interface like USB, an internal device connected through PCIe or an onboard chip connected through a serial bus. In this thesis, an USB audio interface is used, as it is the most common and easiest to set up.
The latency introduced by the audio interface is measured and presented alongside the latency measurements of the full system, to make the results relatable to other setups.

USB

USB audio interfaces transfer audio data to and from the host using individual isochronous USB streams. These streams lock down a certain amount of bandwidth on the USB bus and guarantee that the data is transferred at a constant rate.[7]
Each packet in the stream can contain up to 1024 bytes of audio data. At full speed (USB 1.0), a packet is transferred every 1 ms, and at high speed (USB 2.0) every 125us. At 24bit resolution, each packet can contain 256 samples (24bit resolution is padded to 32bit for USB transmission [8]).
Even at USB 1.0, this is enough throughput to transfer 5 channels of 48kHz audio (48 samples per channel per millisecond). Unfortunately, some audio interfaces only support power of two periods, which do not align well with the USB packet transmission intervals.
For example, at 48kHz and 64 frame period, a new buffer should be ready to process every $\frac{64}{48kHz} = 1.33ms$. This interval does not align with the USB packet transmission intervals of 1ms or 125us, which, especially with USB 1.0, leads to very irregular buffer processing intervals. As a result, the audio processing software has less time to process the audio data before causing a buffer underrun (xrun). With USB 2.0, the misalignment of up to $\sim 80us$ is not as critical, but still not ideal for hard real-time audio use cases.

I2S

I2S is a serial bus protocol based on the I2C protocol, that transfers audio data between chips on a Printed Circuit Board (PCB). Due to its on-chip design and packet-less transfer, I2S introduces much less latency than USB and is, therefore, the preferred choice for real-time audio processing. The latency of an I2S interface depends on two factors (the DSP buffer size is excluded as it has the same effect on I2S and USB interfaces):

- the group delay of the ADCs / DACs
- the transfer latency between the interface chip and the process memory

The group delay is the time it takes for a sample to be ready for transfer at the ADC (or the reverse for the DAC) and is the term to look for in data sheets when selecting low latency I2S interface chips.[9] Since this duration depends on the target sample rate, it is usually given in units of $1/f_s$ seconds. For example, the Cirrus Logic CS42436 ADC at single-speed mode has a group delay of $12/f_s = 255us$ for $f_s = 48kHz$.
The transfer latency depends on the PCB design, audio driver implementation and the transfer clock speed and is likely to be negligible compared to the group delay.

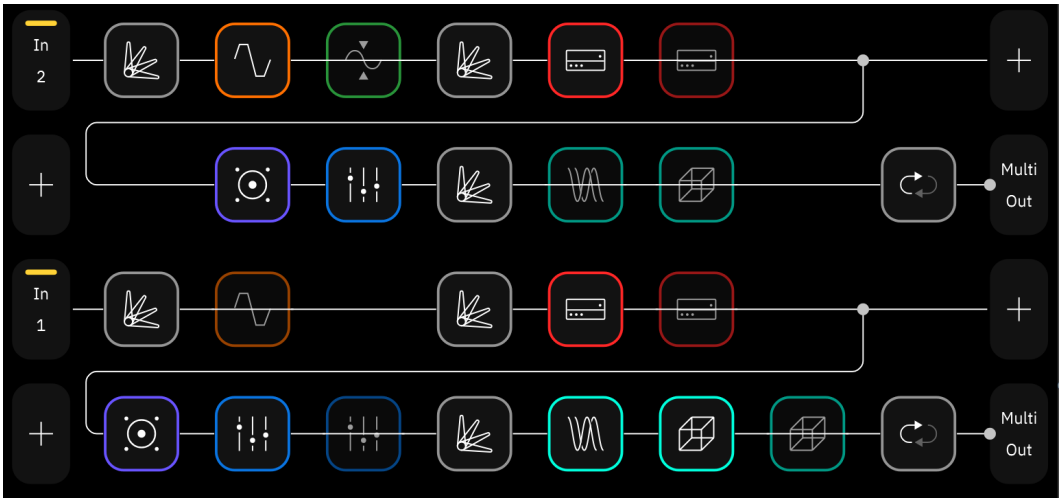


Figure 2.3.: Example of a dual guitar signal chain on the Neural DSP Quad Cortex. Each block represents an audio effect.

2.1.3. Quad Cortex

The Neural DSP Quad Cortex is a digital guitar amplifier and effects processor that runs its audio processing algorithms on a dual-core ARM Cortex-A53 CPU and four SHARC DSPs.
This device has inspired this thesis in many ways and is therefore used to benchmark the results.

The signal flow is divided into 4 paths, each capable of processing 8 audio effects on completely independent signals in either mono or stereo. Lanes 1 and 2, as well as 3 and 4, can be split and mixed together to create parallel signal chains or extended to double the number of effects. Additionally, lanes 1 and 2 can be used as inputs for lanes 3 and 4, creating a maximum serial signal chain of 32 effects. According to Neural DSP, lanes 1 and 2 are processed on one of the ARM cores and lanes 3 and 4 on the other. It is, therefore, recommended that the CPU-heavy audio effects be distributed evenly across the lanes.

The Quad Cortex differentiates between scene and preset switching. When switching scenes, only the parameters of already loaded effects can be changed. As a result, there is no gap in the audio signal. To change loaded effects, the user must switch presets. Since this rebuilds the entire signal chain, there is a significant pause in the audio signal, making this feature unsuitable for in-song effect changes.
A similar distinction between soft parameter updates and breaking parameter changes is required on the Jetson.

The Quad Cortex uses machine learning to create "captures" of guitar amplifiers and pedals. Since the CPU is not suited to train neural networks, this feature is likely a major reason why the four SHARC DSPs were included. Sadly, there is no public information available on whether the SHARC DSPs are used only in training or whether they are also used for inference or offloading other complex audio effects during real-time processing.

Similar to the hardware specifications, limited public information is available about the Quad Cortex's software architecture. However, I was able to gather some insights by measuring the round-trip latencies of different signal chains using an oscilloscope.

The Quad Cortex exhibits a round-trip latency of $\sim 1.9ms$ on a blank preset. However, after adding just a single amplifier or capture block, the latency increases to $\sim 2.2ms$. This indicates that the Quad Cortex has a period size of at most 16 frames and can dynamically adjust the number of processing periods based on the complexity of the audio effects used. Furthermore, it follows that both the amplifier and capture audio effects take longer than 0.3ms to process. Less complex audio effects like a gate or EQ can be processed within a single period.

When there is an audio effect on the third or fourth lane, their outputs are delayed by one period compared to the first and second lanes. This delay could stem from delayed scheduling of the second ARM core or a bottleneck in the output stage. It's surprising that Neural DSP does not synchronize the output of all lanes to prevent phase issues when mixing them together.

The Quad Cortex can process up to 14 captures in series with $\sim 10.5ms$ latency and four in parallel with $\sim 3.5ms$ / $\sim 4.2ms$ latency for lanes 1/2 and 3/4, respectively. The latter is limited by the number of lanes available in the User Interface (UI), not the device's processing power. In contrast, it can only run four amplifiers in series with $\sim 7.2ms$ latency and in parallel with $\sim 3.5ms$ / $\sim 4.2ms$ latency for lanes 1/2 and 3/4, respectively.

The Quad Cortex is equipped with built-in guitar cabinets and light IR loaders. These are implemented so efficiently that it can process up to 20 of them in at $\sim 7.3ms$ latency. However, it can only support a maximum of 8 regular mono IR loaders. The difference in performance between the light and regular IR loaders is likely caused through differing IR lengths. Regrettably I was unable to locate any information on how the built-in guitar cabinets are implemented.

2.2. Software

2.2.1. JetPack

Nvidia is aware of the difficulties developers face when matching the versions of components from the CUDA development ecosystem. To solve this problem, they release collections of compatible soft- and firmware versions under the name JetPack SDK. Its core components include:

- a Jetson Linux or Linux for Tegra (L4T) - an Ubuntu-based Operating System (OS) for the Jetson
- a fitting Driver package
- a compatible CUDA Toolkit

The best way to install JetPack SDKs is by using Nvidia's SDK Manager. It sets up a host environment within a docker container, which can then be used to flash a Jetson development kit. This helps to avoid any version conflicts between the host environment and the Jetson to ensure that remote debugging and profiling tools work as expected. For example, the Nsight Systems Graphical User Interface (GUI) is not installed on the Jetson and must be run on a host machine that is connected to the Jetson via SSH.

In general, using the SDK manager is a great help and will work for most use cases, but it pays to be aware of what versions are required and what is actually installed.
For example, the SDK manager did not install the developer version of the openCV library on the Jetson, which was required to run neural network inference using TensorRT. I then had to manually install the openCV 4.6.0 developer package to achieve compatibility with the Jetson's compute capability of 8.7, even though the installed JetPack 5.1.2 version includes openCV 4.5.4 by default.

2.2.2. CUDA

CUDA is a C++ extension from Nvidia to facilitate parallel execution of algorithms on their GPUs. Running SIMD operations puts high requirements on data access. For this reason, GPUs can only operate data that resides in the GPU's device memory. As a result, when working with CUDA, it is important to distinguish between operations run on the host (CPU) and the device (GPU) or transferring data between them.

All information in the following subsections is based on the CUDA 11.4 programming guide.[10]

Kernel

CUDA extends C++ by allowing the programmer to define kernel functions, that when called, are executed multiple times in parallel, as opposed to only once like regular C++ functions.[10] For this purpose, CUDA introduces new declaration specifiers, such as:

- `__global__` - indicates a function that is called from the host and executed on the device
- `__device__` - indicates a function that is called from the device and executed on the device
- `__host__` - indicates a function that is called from the host and executed on the host

The developer can call these functions using up to four launch arguments enclosed in angle brackets (<<< ... >>>):

- grid dimension n_b - the number of blocks that are launched
- block dimension n_{tpb} - the number of threads per block
- shared memory size - the number of bytes of shared memory that is dynamically allocated per block
- stream - the stream in which the kernel should be executed

The number of times that CUDA executes a kernel is given by $n_t = n_b \cdot n_{tpb}$. Developers may allocate shared memory if threads within a block need to communicate with each other; otherwise, they can leave it at its default value of 0. Streams inform the scheduler about dependencies between launches and are necessary to launch multiple kernels in parallel. For more details about CUDA stream launching, please refer to section 2.2.2.

Understanding a kernel's potential work and the GPU's hardware capabilities is essential when choosing optimal launch arguments. Starting at the lowest level, the scheduler executes work in warps of 32 threads. Therefore, outside of specific use cases where the developer is aware of the imposed limitation, the number of threads per block should always be a multiple of 32 to fully occupy all cores in a single SP.

Blocks are scheduled on a single SM and can execute warps on each of the four SPs within the SM. Therefore, the number of warps per SM should be at least four, but preferably more, so the scheduler can hide memory access latencies through pipelining. Following this, one could argue that kernels should always be launched with the maximum number of threads per block. However, the Jetson has a maximum number of threads per block of 1024 and a maximum number of threads per SM of 1536. Therefore, fully utilising a single SM with a single kernel would require it to be launched with two blocks of 768 threads each (or a variation thereof). Unfortunately, this only works if the kernel is embarrassingly parallel and has no inter-thread dependencies.

Threads can easily communicate with each other within a block, while communication between blocks is possible but challenging. Therefore, algorithms with inter-thread dependencies should be launched within a single block. Finally, kernels should only be launched with as many threads as the algorithm requires. For example, while admittedly a rare concern for developers that use the GPU for its intended purpose, buffer sizes of less than 128 samples in audio processing do not require an entire SM to process.

To summarize, when choosing launch arguments, developers should consider:

- the number of threads the algorithm requires and round it up to the nearest multiple of 32
- whether inter-thread communication is necessary and adjust the number of blocks accordingly
- the max number of threads per SM, which should be split evenly among blocks to achieve optimal SM occupancy. Refer to Section 3.10 for more information.
- other work that may run in parallel and constrain the kernel to as few resources as is practical

This shows that while the CUDA API and GPU architecture allow for amazing cross-compatibility between different GPUs, utilisation may not always be optimal if the kernel launch arguments are not adapted.

Choosing the optimal launch arguments is a complex task many developers strive to automate. The CUDA API provides functions like `cudaOccupancyMaxPotentialBlockSize()` and `cudaOccupancyMaxActiveBlocksPerMultiprocessor()` to query the hardware specifications at runtime. However, these still leave many decisions up to the developer.
Introduced in 2019, KLARAPTOR is a tool built on top of the LLVM Pass Framework and NVIDIA CUPTI API to determine the optimal values of kernel launch parameters dynamically.[11] To my

knowledge, this is the only tool that optimises launch parameters while considering other workloads running on the GPU instead of just maximizing a single kernel's performance.

Persistent Kernel

Audio signal processing, in general, contains a lot of serial dependencies that prevent commands from running in parallel. The IR convolution audio effect described in `Section 4.4.1` is an excellent example of how parallelism must be realized on instruction level within the serial commands. As a result, audio processing depends on executing many, often small, workloads, which is not the use case GPUs were designed for.

A 2017 study by the Institute of Embedded Systems (ZHAW) found that the CUDA scheduler does not handle this type of workload well. [1] While their audio processing loop completed within the real-time constraints most of the time, they encountered intermitted delays that caused xruns in the audio driver. For this reason, they decided to implement the entire audio processing loop inside a single kernel.
This kernel is launched once at the start and then continuously processes audio data in a sample loop from and to fixed memory addresses. They hard-coded all audio effects into the kernel and updated the parameters intermittently through shared memory. While this solved the intermittent launch delays, persistent kernels have their downsides.
Only one thread can be allocated per audio channel to prevent out-of-order execution. As a result, at least 32 channels would need to be processed in parallel to fully utilise a single SP.
Since the study's goal was to run a multi-channel mixing console on the GPU, this was not a problem, but it still led to suboptimal utilisation of the GPU capacity.
Additionally, hardcoding audio effects into a kernel offers little flexibility for dynamic signal chain building.
Lastly, looking at the kernel implementation, which comprises over 1000 lines of code, I dread the difficulties of debugging and maintaining such a solution.

Streams

A CUDA stream is a sequence of commands that execute in order.[10] All kernels and host ↔ device memory copy commands that do not specify a specific stream are launched in to the default stream and are implicitly executed in order. However, according to the API synchronisation behavior described in the CUDA runtime API documentation, this does not mean that they are concurrent with respect to the host.[12] As a result, developers should synchronize the host to the device before accessing the output of GPU operations using either `cudaStreamSynchronize()` to synchronize to a specific stream or `cudaDeviceSynchronize()` to synchronize to the completion of all previously launched work on all streams.
For two commands to run concurrently, they must be launched in different streams. Even then, there are limitations to the overlapping behaviour of commands as described in the section Implicit synchronisation of the CUDA programming guide.[10]. A good rule of thumb to prevent implicit synchronisation is to launch the concurrent commands back-to-back without launching other commands in between.
As the name suggests, streams are helpful for managing process flows with straightforward execution paths and few dependencies between them. However, in my opinion, as soon as the number of execution paths starts varying or dependencies between them become more complex, using the API becomes cumbersome. Furthermore, while the stream guarantees in-order-execution of commands launched into it, it does not perform any launch overhead optimisations. For these reasons, I will primarily focus on using graphs for concurrency management in this thesis.

Graphs

With the release of CUDA 10.0 in 2018, NVIDIA introduced the CUDA graph API. It promises to reduce the launch overhead of repetitive command sequences and allow for better runtime optimisations.
To create a graph, developers can either record their existing stream-based command sequences into a graph, create one from scratch or combine the two options. For the latter the current graph has to be retrieved from the recording stream using `cudaStreamGetCaptureInfo()`, modified and updated on the stream using `cudaStreamUpdateCaptureDependencies()` to continue stream recording. Using this method tends to inflate the code massively.

Creating a graph using stream capture will only record the commands executed on the stream that is being captured. It is possible to capture multiple streams into individual graphs simultaneously, but the developer needs to handle node dependencies between the streams using events (`cudaEventRecord()` and `cudaStreamWaitEvent()`) instead of stream synchronisation. As a result, stream capture is mostly useful for capturing linear workflows.

Creating a graph from scratch is more flexible as the developer can define specific dependencies between all nodes. The dependencies of a node can either be set at creation time or later using `cudaGraphAddDependencies()`. Unfortunately the CUDA graph API is not as intuitive as the stream API and involves a lot of void pointers posing much potential for errors. I recommend to create wrapper classes for the most frequently used node types as described in Section 4.5.

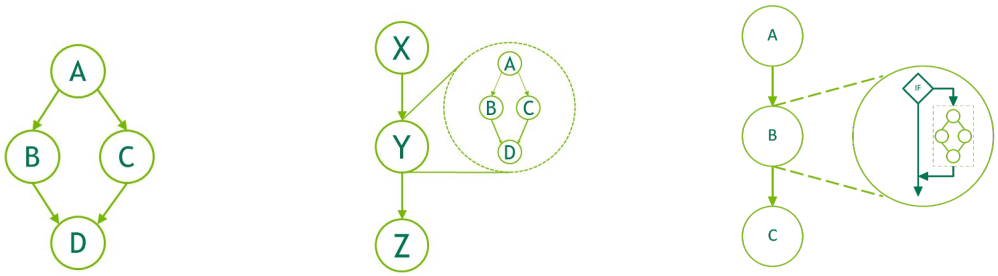
A CUDA graph can contain any of the node types described in the Section Node Types of the CUDA programming guide.[10] However, only the following types are relevant to this thesis:

- **Kernel Node** - a node that launches a kernel
- **Memcpy Node** - a node that copies memory between host and device or within the device
- **Memset Node** - a node that sets a memory region to a specific value
- **Child Graph Node** - any graph can be added as a node to another graph
- **If Node** - a node that only executes if a specific condition is met

Launching a graph is more efficient than launching the same work in a stream because, when launching a graph, CUDA executes all commands precisely the way they were built/recorded. Any pointers and by-value parameters are resolved at build/record time and will not change unless specifically updated. Thanks to this information, the scheduler can plan optimal scheduling of the graph once at instantiation and is fully prepared once the graph is launched.
All this optimisation comes at the cost of the time it takes to instantiate the graph. The net worth of graphs over streams is highly dependent on their complexity and the number of times they are launched. For example, a 2023 study found an improvement of up to 14% for medium-sized workloads, but no difference for large workloads and worse performance for small workloads compared to stream launching.[13] However, readers should keep in mind that they averaged the execution times over only 100 launches, used much larger inputs than are typical in real-time audio, and the used Rodinia[14] benchmark code does not seem very representative of real-time audio processing in regards to the size and complexity of the graphs. Furthermore, due to the high frequency at which real-time audio interfaces execute the process loop, spending milliseconds in preparation to save microseconds in execution time is very much worth it.

It may seem that the rigidity of an instantiated graph is suboptimal for a process that needs to adapt to changing parameters on the fly. While it is true that CUDA needs to rebuild the entire graph when changing the process buffer size and adding or removing audio effects, updating soft parameters like the mix ratio of a convolution is very straightforward.
The fact that launched graphs are immutable is an advantage because it completely eliminates the need for staging areas for soft parameters. The UI can update any node in the graph without affecting an active launch and the graph can be re-launched with the new parameters as soon as the previous

launch has finished. More impactful parameter changes, like exchanging the IR of a convolution node, still require allocating new buffers and staging the new frequency domain representation of the IR.



(a) Example of a simple cuda graph, where the nodes B and C can process in parallel, following the completion of node A.

(b) Visualisation of how a child graph can be added to a parent graph like any other node type.

(c) Example of a child graph that is only executed if a specific condition is met.

Figure 2.4.: CUDA graph examples[10]

Profiling

Nsight Systems can profile large processing loops and generate a visual timeline, showing durations of executed commands, the streams they were executed on and GPU utilisation at the execution time. Additionally, it collects statistical information about command execution times, such as the minimum/maximum, mean, and standard deviation. Note that the JetPack SDK does not install Nsight Systems on the Jetson. Instead, it has to be run on an x86 development machine and connected to the Jetson via SSH. Nvidia recommends running the software on the host Docker image generated by the SDK manager, as this will prevent version conflicts between Nsight Systems and the CUDA toolkit version installed on the Jetson. Be aware that the SDK manager may install two versions of Nsight Systems: one for profiling on x86 and one for Tegra architecture. When starting Nsight Systems from the Command Line Interface (CLI), it may be necessary to rebind the nsys-ui command to use the correct version.

Nsight Compute is a tool that can profile individual kernels to identify optimisation potential. It steps through the program kernel by kernel and provides detailed reports on memory and compute throughput, launch arguments, and occupancy. In my experience, Nsight Compute focuses on maximizing throughput and occupancy without considering the impact on other GPU workloads. As a result, its optimisation suggestions have not been very helpful for real-time audio processing. Similar to Nsight Systems, Nsight Compute is not installed on the Jetson and needs to be run on an x86 development machine.

Compute Sanitizer is a wrapper for multiple runtime tools that can detect memory initialisation/access and thread synchronisation problems.

- **Memcheck** - detects out-of-bounds and misaligned memory access
- **Racecheck** - detects shared memory access race conditions
- **Initcheck** - detects access to uninitialized global memory
- **Synccheck** - detects thread sync problems

Compute Sanitizer is a CLI tool that can be run on the Jetson using the command:
`compute-sanitizer -tool <toolname> <app>`.

2.2.3. Thrust

Thrust is CUDA's version of the C++ Standard Template Library (STL). It enables developers to write functors and apply them to device vectors, similar to how vector operations work in C++. One of its most useful features is the ability to perform vector-reduction operations.
Reducing vector data can be accomplished in a kernel using atomic variables and thread barriers. However, I prefer to delegate thread synchronisation to a library whenever possible.

It is important to note that while Thrust commands can be launched into streams, achieving command-level concurrency is not possible for all of them. For instance, the reduce command (and any other command that returns a left-hand-side value) will implicitly synchronize the host with the device before returning the result. This makes it impossible to launch concurrent commands from a single host thread. To work around this, developers can use operations that store the result in a device vector.
Moreover, prior to version 1.16, all API commands would implicitly synchronize the host to the assigned stream before returning to the host context. Therefore, it may be beneficial to use the CUB library for certain operations when working with earlier versions of the CUDA Toolkit.
The CUB API is slightly more complex, as it is designed to launch work on multiple devices, but it provides similar functionality to Thrust without the concurrency limitations described above. To see an example of using Thrust and CUB in conjunction, refer to Section 4.6.

2.2.4. Jack

Jack is the preferred solution for running real-time audio on Linux. It can work with buffer sizes as low as any USB audio interface can set and provides an easy-to-use C++ API for building audio processing pipelines.
Setting up Jack to run in real-time mode on Ubuntu requires some effort, but the process is well documented on their website.
Jack can be run using either the CLI or the qJackCtl GUI. Since I use shell scripts to start both Jack and the audio processing pipeline simultaneously, I will be using the CLI throughout this thesis.

There are many parameters and options that can be set through the CLI. The most important are:

- **-P** - sets the real-time scheduler priority of the Jack server and should be set to 95.
- **-d** - sets the driver backend to use. On Ubuntu, this should be set to Advanced Linux Sound Architecture (ALSA).
- **-r** - sets the sample rate
- **-p** - sets the period size (to the number of samples per channel (frames) to be processed per period)
- **-n** - sets the number of periods of transfer latency between the interface and process memory (min/default value: 2)
- **-D** - starts the server in duplex mode to provide both capture and playback ports
- **-C** - sets the capture device
- **-P** - sets the playback device

For a full list of options refer to the man page.

In order to understand how the period size p and the number of periods n_p affect the round trip latency, it is important to comprehend how Jack transfers audio data between the audio interface and the process memory.
For each capture and playback port, Jack allocates a ring buffer large enough to contain multiple periods of samples. At any given time, Jack uses one p-sized section to send/receive audio data to/from the audio interface, while the audio processing pipeline uses another p-sized section to read/write audio data. Because of this, n_p must have a minimum value of 2.

Starting Jack with an USB audio interface and $n_p = 2$ will report two periods of latency for both capture and playback. This coincides with the measurements presented in Section 5.1. However, when listing the port latencies using the `jack_lsp -l` command only one period of latency is reported for the capture port.
According to an article on the linux audio wiki, Jack adds one period of latency specifically for USB audio interfaces.[15] Unfortunately, the article does not explain whether this stems from an expanded ring buffer on Jack's side or from a limitation of the USB driver. For the purpose of this thesis, I will assume that the additional period of latency is added through the ring buffer.
I was unable to find any official documentation on how the ring buffer size is calculated. However, based on the step-by-step visualisation presented in Figure 2.5, a ring buffer of size $b = (n_p + 1) \cdot p$ is required to introduce n_p periods of latency per port.

The transmission between the buffer and the audio interface introduces additional latency t_{USB} for both the input and output path.
The round trip latency is therefore calculated using $t_{rt} = 2 \cdot (t_{USB} + n_p \frac{p}{f_S})$. According to Section 5.1, this should already provide the processing pipeline with the entire third period to process the audio data. However, the measurements presented in Section 5.1 show that Jack adds a fifth period of latency for applications that use the Jack C++ API.

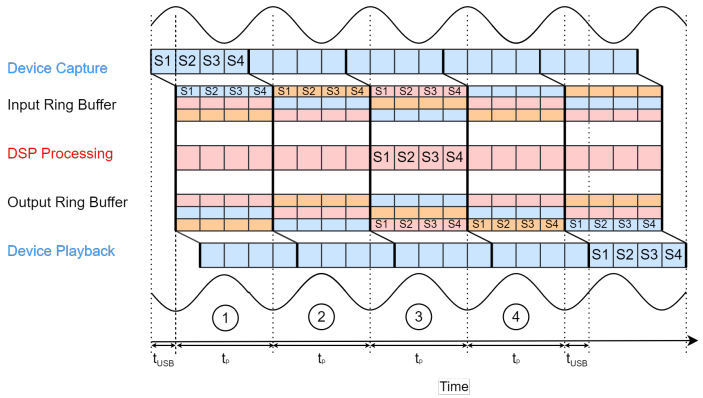


Figure 2.5.: Simplified visualisation of the jack loopback latency. The image shows how a ring buffer of size $b = (n_p + 1) \cdot p$ could introduce $n_p = 2$ periods of latency per port. The location of a period of $p = 4$ samples is tracked over all time steps. The changing colors within the ring buffer signify whether the section is currently being accessed by the audio interface or the DSP pipeline.

2.3. Real Time Audio

Real-time requirements in live audio processing depend on two main factors:

Firstly, the subjective opinion of the artist is essential, as they need to hear the processed audio on stage without any noticeable delay. A study by the Audio Engineering Society (AES) found that most musicians preferred latencies ranging from $1ms$ to $10ms$.[16]

Secondly, physical constraints include the propagation delay between the instruments, the PA and the audience. The speed of sound in air is $343m/s$, which means that sound travels approximately $0.34m/ms$ or takes $\sim 3ms$ to travel one meter. For example, if the audience is 10m away from a snare drum and the PA is hung in the middle between the two, the PA should play the snare drum sound $\sim 5 * 3ms = 15ms$ after the drummer hits it to be in phase.

Usually, the artist's monitoring latency is the more critical constraint, as the propagation delay from the instruments to the audience is typically higher than the maximum monitoring latency musicians can tolerate.

When playing electric guitar using Virtual Studio Technology (VST) plugins as amplifiers, I tend to set the period to 128 frames, which at 48kHz resolves to $\sim 13.5ms$ round-trip latency on a Focusrite Scarlett 8i6 3rd Gen. This period allows for a complex signal chain while still providing a responsive playing experience. I can play with a period of 256 frames, resulting in a round-trip latency of $\sim 27ms$, but the delay is noticeable.

When designing real-time audio processing devices, it is essential to consider that other devices may be connected in a series upstream or downstream.
For example, I use a Quad Cortex in my band to feed analogue signals into a Behringer X32 mixing console. From there, 16 channels are transmitted digitally over Ethernet to a Behringer P16 personal mixer, where I mix my monitor signal. Finally, the stereo IEM signal is transmitted wirelessly to my InEar monitors.
Each of these four devices can introduce a maximum of $2.5ms$ of latency to maintain monitoring latency below 10ms. With the $1.6ms$ hardware latency observed on the Quad Cortex, this leaves just 1ms for audio processing.

It's worth noting that these numbers are not fixed, since even a high-quality commercial product like the Quad Cortex can add up to $\sim 10.5ms$ of latency for complex signal chains. However, keeping the round-trip latency as low as possible is undeniably essential to establishing a new product.

3. Concept Elaboration

This chapter describes the experiments carried out to assess the viability of utilising a GPU for real-time audio processing.

3.1. Overview

This section outlines the scope of the experiments, and provides an overview of the hardware and software setup.

3.1.1. Scope

To assess the capability of a GPUs as a platform for audio processing, the following methodology was employed:

Initially, a loopback test was conducted with the audio data being routed through GPU memory. This step was crucial to verify the functionality of the hardware setup and all associated software components and confirm the GPU's capacity for real-time audio processing.

Subsequently, the study focused on implementing audio effects with varying potential for parallelism. This approach aimed to highlight the GPU's potential in processing audio while demonstrating its ability to handle serial algorithms efficiently.

The third phase involved optimising audio effects. Algorithms were rewritten using different patterns or executed using different APIs to determine the most efficient method for implementing audio effects on the GPU.

The final step was the development of an API that facilitated the combination of audio effects into complex signal graphs. This API enabled the construction and operation of signal graphs that were benchmarked against traditional mixing consoles and digital audio effect modellers, such as the Quad Cortex.

During this process, the software was designed with the following requirements in mind:

It should process real-time audio effects with a period size ranging from 32 to 128 frames. A minimum period of 32 frames (0.67ms at 48kHz) was chosen because smaller periods would lead to suboptimal utilisation of CUDA's 32 thread warps. A maximum period of 128 frames (2.67ms at 48kHz) was chosen as a result of the real-time constraints described in Section 2.3.

The API should offer an interface for audio effects that allows for the uniform implementation of wildly different algorithms. This uniformity ensures that any combination of audio effects can be processed with minimal setup effort.

The API should implement clearly defined setup and teardown phases to facilitate loading and unloading signal graphs at runtime without leaving any allocated memory behind.

All audio effects should implement a soft parameter update method that enables users to change non-breaking parameters like gain or mix-ratio at runtime.

While the last three requirements may seem peripheral to the primary goal of this thesis, they are, in my opinion, crucial for establishing the GPU as a viable audio processing platform in the long term. The results of this thesis will hopefully convince audio developers of the potential of the GPU, but providing an easy-to-use API will be key to actually getting them to use it. Although I do not plan to implement an actual user interface in the scope of this thesis, these are important design considerations that should be followed from the beginning of the project.

3.1.2. Hardware Setup

The hardware setup visualized in Figure 3.1 consists of the following components:

- Nvidia Jetson Orin NX 16GB
- Modular Vision System carrier board
- Focusrite Scarlett 8i6 3rd Gen USB Audio Interface
- Oscilloscope for latency measurements
- Various I/O devices

The audio interface is connected to the Jetson via an USB 3.0 port, but an USB 2.0 port would suffice.

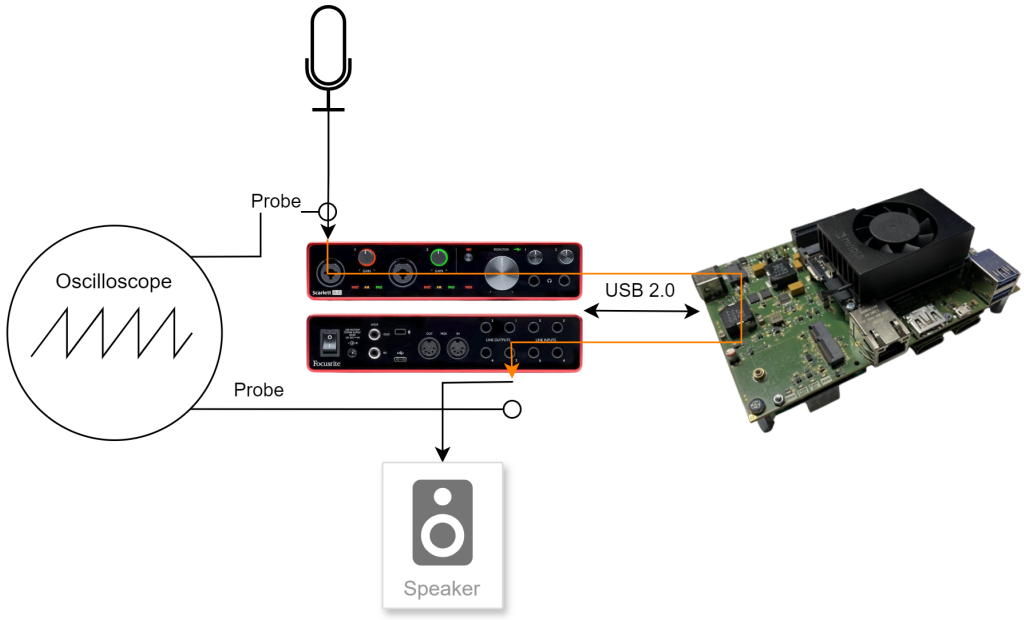


Figure 3.1.: Visualisation of the hardware setup[17][18]

3.1.3. Software Setup

The software setup consists of the following components:

- JetPack 5.1.2
- CUDA Toolkit 11.4
- Jack Audio Connection Kit 1.9.19 (jackd2) including the development library (libjack-jack2-dev)
- CMake 3.21.3
- GCC 8.4.0
- TensorRT 8.0.1
- OpenCV 4.6.0

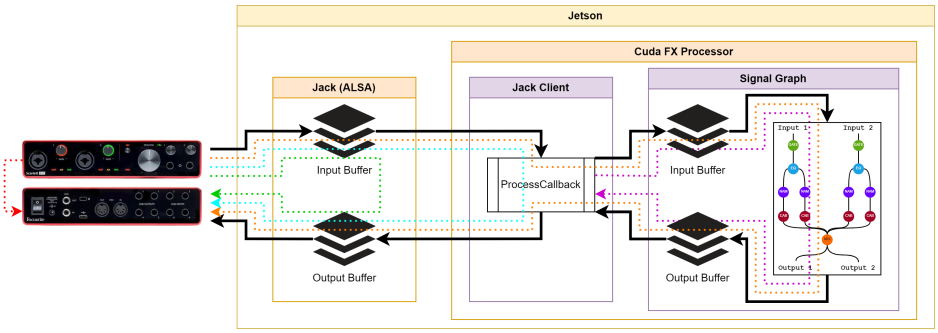


Figure 3.2.: Signal Path Overview[18]

3.1.4. Signal Path Overview

Figure 3.2 shows how the signal is routed through the system. During the experiments, an oscilloscope and the C++ chrono library will be used to measure the latency of different signal paths.

The **Direct Monitoring Latency (red)** measures the round-trip latency of the audio interface using the direct monitoring feature.

The **Jack Loopback Latency (green)** measures the round-trip latency by feeding the input directly back to the output using only the Jack audio server within the Jetson. The impulse is generated using a microphone, and the latency is determined by measuring both the analogue input and output using an oscilloscope.

The **Jack Client Loopback Latency (blue)** measures the round-trip latency of copying the input buffer straight to the output buffer within the Jack client implementation using the same process as the Jack loopback latency.

The **System Round Trip Latency (purple)** measures the round-trip latency of the entire system using the same process as the Jack loopback latency.

The **Signal Graph Processing Latency (orange)** measures the time it takes the signal graph to process a single period using the C++ chrono library.

Measuring different signal paths enables the assessment of latencies introduced by less transparent parts of the system, such as the USB data transfer and the Jack audio server. Additionally, cross-validating the results helps identify potential issues in the measurement process or the signal routing.

3.2. Jack Driver Interface

In order to process real-time audio within C++, I needed to interface my application with a running Jack audio server. For this purpose, I implemented a Jack client that manages all interactions with the Jack server, keeping the dependency on the Jack library contained within a single class. As this part of the software acts like an audio driver towards the other components, I will now refer to it as the Jack driver.
The primary purpose of the Jack driver is to register a process callback method with the Jack server. Once the driver is connected to the Jack server, it will call this method whenever new audio data is available. These callbacks should arrive regularly at intervals that depend on the sample rate and period size. For example, with a sample rate of 48kHz and a period size of 64 samples, the method should be called every 1.33ms. However, when I validated this using the chrono library, I found that the callback was called in a repeating irregular pattern of approximately 2ms, 1ms, and 0.01ms.
After researching the issue online, I found several posts suggesting that USB audio interfaces work best with period sizes that convert to a multiple of 1ms duration. [1] There I learned that the USB frame transmission interval is 1 ms for USB 1.0 (full-speed) and 125 ms for USB 2.0 (high-speed). Since the Behringer Uphoria UM2 interface I used was only USB 1.0 capable, there was a significant mismatch between the USB frame transmission interval and the expected process callback interval.

Using the same interface with a period of 48 frames would have changed the callback interval to $1ms$ and resolved the issue. However, for kernels that process one sample/frame per thread, a period of 48 frames results in half-full warps being scheduled. This thesis aims to test the limits of a GPU by running as many audio effects in parallel as possible. Therefore, scheduling half-full warps should be avoided.
Fortunately, I was able to replace the Behringer Uphoria UM2 with a Focusrite Scarlett 8i6 3rd Gen interface. With this USB 2.0 capable interface, I could run the Jack driver with a period of 64 samples and a sample rate of 48kHz without encountering any issues.

3.3. Impulse Response Convolution

When I began this project, I was lucky to come across the "cuda-audio" git repository by the user Limitz, which contains a CUDA implementation of a real-time convolution reverb effect. His code has been immensely helpful in getting me started with the CUDA API. However, it was also a source of great confusion since Limitz seems to be a follower of the "the code is its own documentation" philosophy.
The original implementation included a MIDI interface for adjusting parameters like pre-delay and mix ratio and even modifying the impulse response. However, these features made the code more complex and made it harder to comprehend the convolution algorithm. As a result, I chose to remove all non-essential code and focus on the core elements.
Next, I needed clarification on why Limitz would copy the stereo input channels into a complex buffer, perform a Complex-to-Complex (C2C) Fast Fourier Transform (FFT) on the buffer, and then unpack the result into two separate complex buffers. After numerous attempts to rephrase my Google search queries, I finally found an archived website that explained the process.[19]
The symmetric property that allows algorithms to discard half of the FFT result of a single real signal can also be used to perform an FFT on a complex buffer where the real part contains one channel, and the imaginary part contains the other. If both channels are then convolved with the same impulse response, the result can be transformed back to the time domain using a single C2C Inverse Fast Fourier Transform (IFFT) pass.
It is not even necessary to unpack the complex buffer since the `cufftComplex` data type is a `struct` comprising `float` values, equivalent to an interleaved buffer with two channels. I have implemented this exact process in the `FxConvFd2c1` audio effect, and the code is so compact for what it does that I

[1] List of optimal Jack period sizes
Another list of optimal Jack period sizes
Ardour forum post blaming USB request blocks for the issue

might actually frame it and hang it on my wall. For reference, the process method is added as a code snipped in Section A.1.

Most IRs used to add reverb to an audio signal are in stereo. Due to the difference between the two IR channels, the symmetric property does not hold for the complex multiplication and transformation back into the time domain. Despite the need to unpack the complex buffer and perform the multiplication and IFFT on both channels separately, reducing the number of FFT passes by one is worthwhile. This process is implemented in the `FxConvFd2c2` audio effect.

Finally, I implemented a third variation for single-channel convolution using the real-to-complex FFT and complex-to-real IFFT. This was the approach I planned to use if not for Limitz's implementation. A comparison of the execution times for these three variations is presented Section 5.5.

Before continuing, I would like to point out two important details.
Firstly, guitar cabinet IRs are usually normalized to close to 0dB peak, which causes the convolution result to clip for almost any input signal.
Secondly, the cuFFT API does not normalize the IFFT result and leaves it to the developer to rescale the convolved signal.
So, if anyone attempting to implement this audio effect is wondering why the frequency spectrum of the convolution output is a solid brick of white, it might just be clipping.

3.4. Execution Time irregularities

The first real-time performance test of the `FxConvFd2c2` started out promising. From previsous measurements, I knew it would take less than $500us$ to process an IR of 2^{17} frames. As a result, I configured the period size to 64 frames ($1.33ms$) and started the real-time test. For approximately 30 seconds, the test was running well when the Jack backend suddenly reported the first xruns, indicating an issue. Unfortunately, upon further investigation using Nsight Systems, the identified issue did not yield significant insights.
The problem with Nsight Systems and most other CUDA profiling tools I used during this thesis is that they are designed to analyse large-scale kernels that run on larger data sets and utilise as many threads and blocks as possible. However, the stereo convolution audio effect operates differently, launching up to 15 distinct commands every $\sim 1.3ms$, each running for less than one millisecond.
While using Nsight Systems, I identified commands that took longer than anticipated, with the worst cases coinciding with Nsight Systems' reported profiling overhead on the timeline. Consequently, once I discovered other outliers with no profiling overhead running in parallel, I was uncertain whether this was a natural occurrence or a result of hidden profiling overhead. Moreover, the irregularities did not follow any discernible pattern. They were not constrained to a type of operation, did not occur at regular intervals, and Nsight Systems did not report anything that could have caused the delay. At this point, I was convinced that these must be the same launch irregularities that were encountered in the 2017 study.[1] However, I had already decided not to use persistent kernels due to their limitations described in Section 2.2.2. Therefore, I opted to look for solutions outside of Nsight Systems.

As the first step, I ran the aptly named jfloorit.sh script by Limitz. It optimises GPU performance by forcing all cores to run at maximum frequency using Nvidia's `jetson_clocks` shell script and enabling the USB-FS to utilise up to 1GB of buffer memory. Additionally, I isolated four CPU cores by adding `isolcpus=4,5,6,7` to the primary bootloader configuration in `extlinux.conf` and assigned my application to these cores using the `taskset` command.

In a second step, I attempted to prevent other processes from launching work on the GPU. CUDA can operate in three compute modes: Default, exclusive and prohibited. Exclusive would allow only a single CUDA context to run on the device. Assuming that the irregularities were caused by a scheduling conflict, locking any other context out of the device could solve the issue. Unfortunately, the only tools that can change the compute mode are nvidia-smi and the `nvml` library, which are un-

available for tegra architecture. As a result, I was left to manually killed any process that is expected to use the GPU. On a fresh Jetson Linux install the GUI was the only such process. Fortunately, I was developing remotely from my main computer using an SSH connection to the Jetson. Therefore, deactivating the GUI did not affect my workflow.

Finally, I decided to tackle the None-Realtime-Mode warning that Jack was giving at every start. There is much conflicting information about modifying cgroup settings to enable real-time mode for Jack. Initially, I spent days trying to get rid of the warning but eventually settled on being able to run Jack at a 64-frame period without issue and chose to ignore After exhausting all other options and revisiting the topic, I realized I had been using the deprecated Jack1 API the entire time.
I was confident that I had installed the `jackd2` apt package, so it took me some time to figure out what went wrong. In order to access the Jack API from C++, I had to install the Jack development library. Unfortunately, I installed libjack-dev instead of libjack-jack2-dev. During the installation process, it detected the incompatibility with jackd2 and replaced it with jackd1. I am aware that I had to confirm this action and did so without carefully reading the prompt. But do you read all your apt install prompts?

After reinstalling jackd2, the None-Realtime-Mode warning disappeared. However, none of these measures had a significant effect on the irregularities. The maximum execution time and standard deviation measurements of experiments using all the described measures were so different that it was impossible to ascertain whether they had any effect. At this point, I gave up on solving the irregularities and decided to work around them.

Examples of the irregularities are presented in the Sections 5.2 and Section 5.7.

3.5. Asynchronous Signal Processing

The most effective way to communicate between two processes with uncooperative scheduling is to run them in separate threads connected through a ring buffer. Coming from an IT background where busy waiting is frowned upon, I initially implemented a blocking buffer using mutex and condition variables. This ensured that both threads would yield until notified by data availability, keeping the CPU load at a minimum.
Unfortunately, the CUDA scheduler does not perform well when launching work from a thread that recently woke up from sleep. According to a post on Nvidia's developer forum, it is well known that putting a thread to sleep may impact subsequent CUDA activity. At this point, I was tired of trying to find elegant solutions to peculiar GPU behavior and decided to simply keep the thread spinning while waiting for new data to arrive.
Although using a non-blocking buffer and spinning to keep the GPU thread awake did not resolve the irregularities, increasing the period size was enough to mask them from the Jack backend. Unfortunately, this introduced more latency to the processing loop. However, maintaining a stable audio output was more crucial than meeting the strictest real-time requirements.
Another advantage of decoupling the Jack and GPU threads was the ability to adjust the ring buffer size to match the complexity of the signal chain, similar to what I assume the Quad Cortex does.

3.6. Neural Network Inference

The Neural Amp Modeler (NAM) is one of many VST plugins that utilise neural networks to replicate the sound of guitar amplifiers. However, it stands out in two ways.
Firstly, it is a solo project by Steve Atkinson and still manages to compete with the major players in the market.
Secondly, the VST C++ code and the Python training framework are open source. As a guitarist, I could not think of a better way to showcase the Jetson's real-time neural network inference capabilities.
In his VST plugin, Steve Atkinson uses the Eigen C++ API to calculate the forward pass of his neural amp models manually. This approach is likely the most efficient way to infer his model, and the Eigen API does support GPU execution. However, translating such a complex algorithm into CUDA would have taken much more time than I was willing to spend on a single audio effect implementation. Thankfully, a coworker suggested using the TensorRT API to do the heavy lifting of the inference.

To infer a neural network with TensorRT, I first needed to export a neural amp model to the Open Neural Network Exchange (ONNX) format. Therefore, the first step was understanding the Python training framework and determining where to add an ONNX export function. The models are built using the PyTorch API, and the training process is managed using a PyTorch Lightning wrapper module.
This hierarchy and some of Steve Atkinson's design choices made it difficult to find the right location to add the export function. As a result, I exported my first ONNX model from the PyTorch Lightning module, which generated an ONNX graph containing the entire network in a single node.

TensorRT has stringent limitations on the operations it can interpret and could not parse the graph. At the time, I was unaware that the problem lay with the network being hidden in a functional node, so I started stripping the Convolutional Neural Network (ConvNet) for any operation I thought might pose a problem. Finding no success with that approach, I started looking for other ways to export the model and realised my mistake in exporting the entire PyTorch Lightning module. However, using the `torch.onnx.export` function on the actual PyTorch model unfortunately still parsed the entire network into a single node.
To rule out any unknown variables, I built a basic ConvNet framework from scratch and exported it to ONNX. Finally, TensorRT was able to parse the ONNX graph.

With the help of another coworker and ChatGPT, I was able to identify the `torch.nn.Sequential` module that was used to build the NAM ConvNet as the cause of the issue. `Sequential` is a container module that automatically handles the forward pass sequence for all added layers in series.
Unfortunately, the ONNX export function does not unpack the `Sequential` module but treats it as a single node. After modifying the code to execute the forward pass without the `Sequential` module, the ONNX export function generated a graph that TensorRT correctly parsed.
With a newly trained model, I could finally perform a single pass inference of the NAM ConvNet on the Jetson. The result sounded terrible, but that is to be expected from an amplifier's output before convolution with cabinet IR.
There were still two things that needed to be added to run the inference in real time. Firstly, the ONNX model was exported with fixed dimensions meant to infer a large audio file in a single pass. Secondly, I had to figure out how to feed the real-time buffers into the network to get a continuous output.

For the first problem, a solution presented itself in form of the `torch.onnx.dynamo_export` function. This function promises to export a model with dynamic dimensions, allowing the audio effect to work with multiple buffer sizes without needing a separate ONNX model version for each possible buffer size. Unfortunately, this export function generated functional nodes for a layer that increases the dimensions of the input tensor.
After unsuccessfully replacing the original array slicing operation with `torch.unsqueeze`, the export finally generated a standard node for the `torch.reshape` operation. It is unclear to me how those three operations differ under the hood of PyTorch, but since ONNX exports are a means to an end in this thesis, I refrained from investigating this further.

Figure 3.3 summarizes the different export options and their results.

To solve the second problem, I spent some time analysing how the NAM ConvNet works.
Its main components are two sequentially stacked dilated convolutional networks. The increasing dilation is used to expand the network's receptive field without increasing the number of parameters. This is crucial for audio processing, as predicting new samples heavily relies on past samples [20].
Each dilated convolutional network consists of twelve 1D convolutional layers with dilation values ranging from 1 to 2048, in powers of two. This results in a receptive field of $R = 2 \cdot 2^{12} = 8192$[21]. This means that the network requires 8192 past input samples to generate the next output sample.

For real-time processing, the size of the input tensor needs to be $L = R - 1 + p$, where p represents the period size. While subtracting one sample from the receptive field is optional, not doing so results in the first sample being predicted from complete silence, which is a waste of computation.
During processing, the input buffer is copied into the rightmost part of the input tensor. As a result, the first samples are generated from a history of mostly silence. In successive iterations, the input tensor is shifted to the left before the new input buffer is copied into the rightmost part. This ensures that the network always has a history of the last 8192 samples and that the output will not have discontinuities between period boundaries.

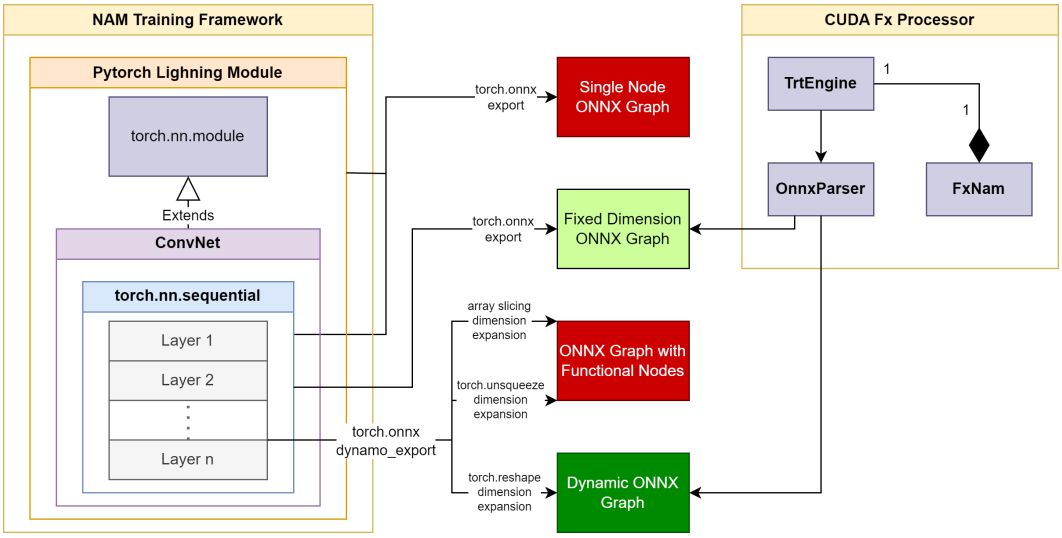


Figure 3.3.: Visualisation of the different ONNX export options and their results. The red marked results can not be parsed by TensorRT. The green marked result can be parsed by TensorRT. However, the export with fixed dimensions is impractical as the input size is dependent on the period size p.

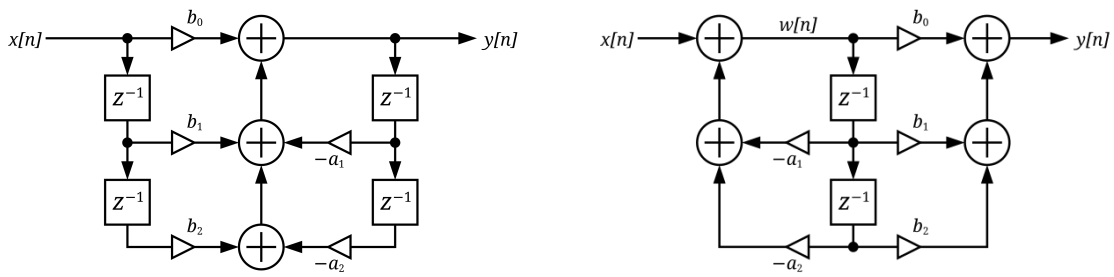
3.7. Implementing an Equalizer

I considered creating a basic graphic equaliser using frequency domain multiplication. However, compared to stereo convolution with seconds-long IRs, adding a frequency-domain equaliser would have added nothing new. With this audio effect, I aimed to explore whether an algorithm with recursive dependencies could be adjusted to the GPU's parallel nature.

Biquad filters are a common method for equalising audio signals. In its Direct Form 1 shown in Figure 3.4a, the biquad filter consists of a second-order Finite Impulse Response (FIR) filter followed by a second-order Infinite Impulse Response (IIR) filter.
The IIR part has a recursive feedback loop that depends on previous output samples. Additionally, multiple biquad filters in Direct Form must be applied to a single audio signal in series. Therefore, a parametric equaliser implementation would consist of multiple cascaded filters, which offer little potential for parallelism.

According to [22], higher-order IIR filters can be split into a series of first and second-order filters to enhance numerical stability. More importantly, this process can also be reversed, allowing multiple first- and second-order filters to be combined into a single higher-order filter.
Subsequently, the higher-order IIR filter can be split into a set of parallel first and second-order filters.[23] This means it should be possible to consolidate a series of biquad filters into a single higher-order filter and then split it into a set of parallel filters to run efficiently on the GPU.

Transforming an IIR filter into its parallel form requires calculating its poles and residuals using Partial Fraction Decomposition (PFD).[23] Matlab and the Scipy Python library offer a function called `residuez` for precisely this purpose. Unfortunately, I was unable to find an implementation in C++, so I would have had to implement a PFE algorithm myself to use parallel form IIR filters.
Similar to Steve Atkinson's Eigen neural network inference implementation, this had the potential to become a very time-consuming task without being integral to the goal of this thesis. For this reason, I decided to settle for sequential biquad filters for my FxEq audio effect.
However, I still managed to introduce some parallelism in the FIR sections and reduce the number of required history buffers using Direct Form 2, so it is not entirely unoptimised. The implementation is presented in Section 4.4.3.



(a) Direct Form 1.[24] (b) Direct Form 2.[24]

Figure 3.4.: Visualisation of a biquad filter.

3.8. Evaluation

Up until this point, I have been testing the audio effects using my eyes and ears. With the amount of refactoring and optimisation that was to come, a way to automatically evaluate the accuracy of the audio effects was required.
The evaluation method needed to fulfil three requirements:

- It has to be able to score the similarity of two audio signals.
- The algorithm needs to be able to handle potential time offsets between two signals.
- The algorithm needs to quantify the time offset between two signals.

A common problem in frame-based audio processing is that the output of an audio effect can be shifted in time. This may occur by design or by accident. Regardless of the cause, an algorithm that can not compensate for phase shifts or propagation delays may falsely report a low similarity score between two signals that are identical, barring the time offset.

Using cross-correlation would cover the first two requirements. Calculating the sliding dot product of two similar signals will produce a high correlation value regardless of the time offset. Sadly, this method does not provide a way to quantify the time offset.
For this reason, I decided to use the Root Mean Square Deviation (RMSD) algorithm shown in Equation 3.1 to evaluate the audio effects. It produces a quantifiable score that punishes larger deviations more than smaller ones and is much cheaper to calculate than cross-correlation. As a result, it can be run for a few thousand offsets to find the best match.

Calculating the RMSD between two signals has excellent potential for parallelism, especially when multiple offsets are employed to detect potential time shifts. Using the Thrust API, I quickly implemented an RMSD calculator that could handle multiple offsets and channels. However, due to the issues with command-level parallelism described in Section 2.2.3, all reductions were run sequentially. Fortunately, the CUB API, which is usually used to distribute work over multiple devices, does not suffer from the same issue and can perform vector and reduction operations similar to Thrust.

Finally, I packed all the boilerplate code for the signal comparison into a class called `Evaluator` which declares the pure virtual methods `setup`, `test`, `process`, `postProcess` and `teardown`. This way, I could derive specific implementations like the `GpuFxEvaluator` from the `Evaluator` class and inject the `GpuFx`-specific code required for testing by overriding the pure virtual methods.
In addition to the `testAccuracy` method, I implemented the `measurePerformance` method to collect execution times over multiple process cycles and return statistical results.

$$\mathrm{RMSD} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2} \tag{3.1}$$

3.9. Vector Memory Instructions

The CUDA global memory instructions support reading and writing words of 1, 2, 4, 8, and 16 bytes.[10] As a result, a single read or write operation can process up to 4 float values. To reduce memory instruction overhead, CUDA extends C++ with vector data types like float2, float3, and float4.

Using these types sets two requirements for the implementation of the audio effects. Firstly, the samples of the different audio channels in the process buffer should be interleaved into a single contiguous buffer so that each vector element corresponds to a single channel sample. This is necessary so kernel threads can perform a single operation on all vector elements using a single memory instruction. Secondly, because the vector data types are implemented as structs rather than actual vectors, all kernel implementations that use them are fixed to a specific number of channels. This is not a problem for audio effects like the `FxConvFd2c1`, as the optimisations only work on a fixed number of channels. Furthermore, contiguous one-dimensional buffers are much easier to handle than segmented multi-dimensional buffers. However, some audio effects, like the equalizer, would benefit from being able to handle a variable number of channels.

For this reason, I decided to test the performance of three variations of an identity kernel that copies the input buffer to the output buffer. The kernel is intentionally simple to ensure that the memory instruction overhead is the dominant factor in the execution time.

- The element kernel uses simple float pointers as parameters and copies only a single value per thread.
- The vector kernel uses float4 pointers as parameters and copies four values per thread.
- The dynamic kernel uses simple float pointers as parameters but also receives the number of channels as an argument. As a result, it contains a nested loop within the thread loop to iterate over the channels and perform a single-value copy operation on each channel. This implementation is included to see whether the CUDA optimiser can infer vector operations from the nested loop.

For a buffer with f frames and c channels, the element kernel is launched $n_t = f \cdot c$ times, while the vector and dynamic kernels are launched $n_t = f$ times to process the entire buffer. All tests were performed using $c = 4$ channels to exacerbate any memory load performance differences between the implementations.

In the first step, I performed grid-search experiments to find the optimal launch arguments for fixed buffer sizes. This test showed no significant performance increases for launch arguments higher than $n_b = 1$ and $n_{tpb} = n_t$ regardless of the buffer size ($f \in \{16, 32, ..., 256\}$). The buffer size range was chosen to cover more than the range specified in Section 3.1.1 to ensure that no performance differences were missed due to the defined real-time requirements.
In a second step, I performed grid-search experiments to see how each kernel handles the different buffer sizes at fixed thread counts $n_{tpb} \in \{32, 256\}$ for the dynamic and vector kernels and $n_{tpb} \in \{32, 128, 256, 768\}$ for the element kernel. Due to the higher launch count n_t of the element kernel, the thread counts $n_{tpb} \in \{128, 768\}$ should be used for a fair comparison with the other two kernels. I decided to include the thread counts $n_{tpb} \in \{32, 256\}$ to see if the element kernel is still a candidate if GPU resources are scarce.

The device buffers were sized to contain the entire memory transferred by all measured executions. This ensured that no two kernel executions used the same memory, preventing data caching from influencing the results. Additionally, the buffer slices passed to the kernel were selected using a strided offset to prevent the GPU from coalescing memory access.
The results of these experiments are presented in Section 5.3.

The dynamic kernel performed the worst through all metrics. Therefore, I assume that the expected optimisation did not occur. The other two implementations were closely matched when comparing them using the fair thread count match. When compared to the vector kernel with the same thread

count, the element kernel performed worse but not significantly so.
Based on this information, I used the element kernels for all audio effects requiring variable channel counts.

3.10. Kernel Launch Arguments

Nvidia recommends using Nsight Compute or relevant CUDA API methods to find the best launch arguments for kernels. However, I found that both options suggest using as many blocks and threads as possible for single kernels without considering other workloads that may need to run in parallel. Consequenty, I used other ways to determine the best kernel launch arguments for my audio effects kernels.

First, I conducted a grid search of reasonable launch arguments. Then I measured the execution times using the `Evaluator`. Finally, I analyzed resource utilisation using the Nsight Systems profiler.

Unfortunately, the irregularities in the results led to large variations in the mean and standard deviation of the execution times, even for subsequent measurements using the same arguments. This made it difficult to draw any meaningful conclusions from the results.
To address this, I refactored the `Evaluator` to calculate curated mean and standard deviation values after clipping the execution times to a constant value in addition to the original statistical results. Clipping the extreme outliers instead of erasing them allows the curated mean and standard deviation to reflect excessive outliers without inducing too much variance. The improved stability made them much more useful for comparing the execution times between different kernel launch arguments.

The kernels of the audio effects either run on small buffers based on the period size or, in the case of the convolution effect, on large buffers based on the IR length.
Focusing first on the former, a grid search for launch arguments at fixed buffer sizes reinforced the conclusion of the previous section: that at such low buffer sizes, the best launch arguments are $n_b = 1$ and $n_{tpb} = f$ or $n_tpb = f \cdot c$ epending on the kernel type.
Additionally, I performed another grid search for buffer sizes $f \in \{4, 8, ..., 256\}$.
Since CUDA schedules work in warps of 32 threads, and these kernels either process one sample or one frame per thread, I wanted to see if the execution times would improve if the buffer size was a multiple of 32. The results of this experiment were equal parts good and bad.

On the one hand, the fact that the execution times are essentially constant for all buffer sizes means that there is no need to restrict the supported period sizes. On the other hand, seeing virtually no difference in execution times between buffer sizes of 4 and 256 samples starkly highlights how far below the intended data size of a GPU these kernels are operating at.
Therefore, I was relieved to see that the kernels of the convolution effect at least showed some performance differences between different buffer sizes.

The following criteria constrain the buffer sizes on which the convolution kernels operate:

- Guitar cabinet IRs tend to be $\sim 100ms$ (4800 frames) long.
- The buffer size should be a power of two to allow for efficient FFT operations.
- The audio effect should be able to process convolutions with IRs of up to 1s (48000 samples) duration.

Based on these criteria, I chose to run launch argument grid-search experiments for the buffer sizes $b \in \{4096, 8192, 16384, 32768, 65536\}$.

From the resulting measurements, it quickly became apparent that maximizing the number of threads per block n_{tpb} yields the best performance. Increasing the number of blocks showed significant benefits up to $n_b = 5$. However, the goal of this experiment is optimal overall resource allocation and not maximum single kernel throughput. As explained in Section 2.2.2, the Jetson has a limit of 1536 threads

per SM. Therefore, launching a kernel with multiple blocks of 1024 threads could cause suboptimal resource allocation.

This assumption is based on the output of the CUDA API function `cudaOccupancyMaxPotentialBlockSize`. While the name may indicate otherwise, the function is designed to return the number of blocks and threads per block that would achieve the maximum potential occupancy for a kernel.[12]

On the Jetson, the function returned n_b and n_{tpb} values of 2 and 768 for almost any kernel I used it on. For this reason, I decided to run the convolution kernels with the launch arguments $n_b = 4$ and $n_{tpb} = 768$ instead of $n_b = 5$ and $n_{tpb} = 1024$ as launch arguments to hopefully constrain them to at most two SMs.

The detailed measurements of this experiment are presented in Section 5.4.

3.11. CUDA Stream vs Graph

Launching work into streams is much simpler than building/recording a graph, instantiating, and finally executing it. Furthermore, the commands comprising the audio effects in this thesis have manly serial dependencies and do not require complex dependency management. Therefore, I utilised the stream API as much as possible to implement the audio effects. One significant advantage of this approach is that all audio effects can run directly on a stream or be recorded into a graph and then run on a stream. The process of switching between the two execution methods is described in Section 4.4.
An issue arose when I needed to update audio effects' source and destination buffer pointers to accommodate the changing pointers retrieved from a ring buffer. Since the CUDA graphs use the exact pointers for kernel and memory operations that are passed to them during recording, I had to find a way to update the pointers of the audio effects in the graph.
This can only be done using a reference to the node that was created when adding a command to the graph. I needed to either switch to manual graph building or use the combination approach described in Section 2.2.2 to obtain references to the graph nodes. Determined to maintain the interoperability between stream and graph execution, I decided to use the latter method.

After checking the capture status of the current stream, the audio effects either launch the command directly into the stream or retrieve the graph from it, add the command as a node, and finally update the stream with the updated graph. Although this method is only required for CUDA commands that whose source and destination pointers change during runtime, it still produces a lot of redundant code and drastically reduces readability.
With some effort, I managed to encapsulate the verbose graph API code in node wrapper classes and reduce the entire procedure to a single line of code. The few parallelised sections, like multi-channel buffer packing and splitting, are run using multi-command node wrapper classes that hide the boilerplate code required for command-level parallelism.

As a result, was able to compare stream vs graph execution times for all audio effects without having to maintain multiple variations of the same code. The latter approach would have taken less time, and based on the results, I will likely drop stream execution compatibility in the future. However, I used the node wrapper classes in many other ways and learned a lot about the CUDA graph API while designing them, so it was worth the effort.
The implementation is described in Section 4.5. The results of the comparison and the conclusions thereof are presented in Section 5.6.

3.12. Audio Effect Execution Time Irregularities

With the informed decision to proceed using the CUDA graph API made, it was time to test the audio effects in real-time instead of using the `Evaluator` class. Unfortunately, the first results were far from satisfactory.
Audio effects that were previously completed in less than 100us suddenly caused xruns when executed in real-time at a period size of 32 frames (667us). When the real-time measurements were compared with the results of the previous experiment, all mean execution times were at least 50% higher.
By that point, I was already aware that the first few measurements in a session tend to be inaccurate because the GPU would still be warming up. Remembering the CUDA scheduler's poor performance after waking a thread from sleep, I assumed that the time between process cycles was too long, causing the scheduler to "fall asleep".
Therefore, instead of spinning only on the CPU during wait times, I implemented a spin kernel to keep the CUDA scheduler "awake". The resulting real-time execution times closely matched the measurements from the previous experiment. Unfortunately, the solution was not yet watertight.

On the one hand, making the spinning kernel too small negatively impacted the performance of the subsequent audio effect kernels.
On the other hand, making the spinning kernel too large delayed the start of subsequent audio effect kernels.
For a perfect solution, I would need to use CUDA events to terminate the spinning kernel whenever new data arrives from the driver. However, at that time, the main priority was to get complex signal chain processing to work. For this reason, I postponed this optimisation and extended the `Evaluator` to simulate real-time periods using `chrono` and the previously developed spin kernel. Since the `Evaluator` does not have to conform to processing intervals of an audio driver, there is no downside from the kernel spinning longer than the actual wait time.

For a detailed comparison of the back-to-back launching, CPU spinning and GPU spinning result, please refer to Section 5.7.

3.13. Signal Graph Execution

Up to this point, I have been using a simple list to combine audio effects into a stereo signal chain and process each sequentially in a for loop using in-place processing on a single buffer. This approach was very efficient and easy to implement, but offered no potential for complex signal routing or parallel execution of audio effects. To showcase how well the CUDA graph API is suited to represent audio processing pipelines, I needed to implement a better API to allow for more complex dependencies between audio effects. For this reason, I decided to copy the CUDA graph API concept to build a dependency graph out of audio effects. The implementation is detailed in Section 4.3.

3.13.1. Buffer Management

The biggest challenge in this endeavour was to find a way to handle the different buffer requirements of the audio effects uniformly.
Multi-channel audio data can be stored in three ways:
A **planar**, a.k.a row-major buffer, stores the data channel by channel in a single contiguous memory block. This is the default layout when allocating fixed-sized two-dimensional arrays in C++.

An **interleaved**, a.k.a column-major buffer, stores the data frame by frame in a single contiguous memory block. This is the layout used to store audio data in PCM files.

Lastly, a **segmented** buffer stores the data of each channel in a separate memory block. This is how the Jack audio backend provides the audio data to the signal graph.

Most audio effects in this thesis require interleaved buffers as input and output. However, to enable complex routing within the signal graph, audio effects like the `FxInputMap` or `FxMixSegment` require a segmented buffer as input so that they can merge the output of multiple vertices into a single interleaved buffer. For more information about these audio effects, please refer to Section 4.4.
At the start of this thesis, I followed Nvidia's example of using void pointers and letting the audio effects cast them to the correct type. However, this approach is error-prone, making debugging the signal graphs' autonomous buffer allocation strategy very time-consuming.

To simplify the buffer handling, I decided to disregard **segmented** as a memory layout and handle every buffer as a collection of memory segments. For this purpose, I created the `Buffer` and `BufferSegment` classes.
A `BufferSegment` contains a single continuos memory block of **interleaved** or **planar** data. A `Buffer` contains a list of `BufferSegment` pointers and a list of pointers to the start of each segment. Additionally the `BufferSpecs` and `BufferSegmentSpecs` classes are used to keep track of the layout and size of the segments.
During configuration, each audio effect defines the `BufferSpecs` it requires for the given buffer size and channel count. This makes it much easier for the signal graph to allocate the correct buffers for each audio effect.
During a process cycle, the audio effects read and write the number of `BufferSegments` they require from the source and destination `Buffers`. Additionally, having the layout and size specifications attached to the `Buffer` allows for dynamic behavior depending amount or size of the segments.

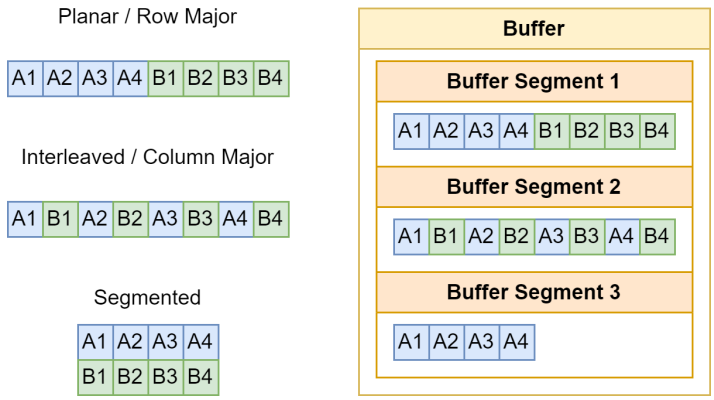


Figure 3.5.: Visualisation of the the different multi-channel data layouts for a simplified buffer with $c = 2$ channels and $f = 4$ frames. The custom `Buffer` class is able to hold `BufferSegments` of differing layouts and channels sizes.

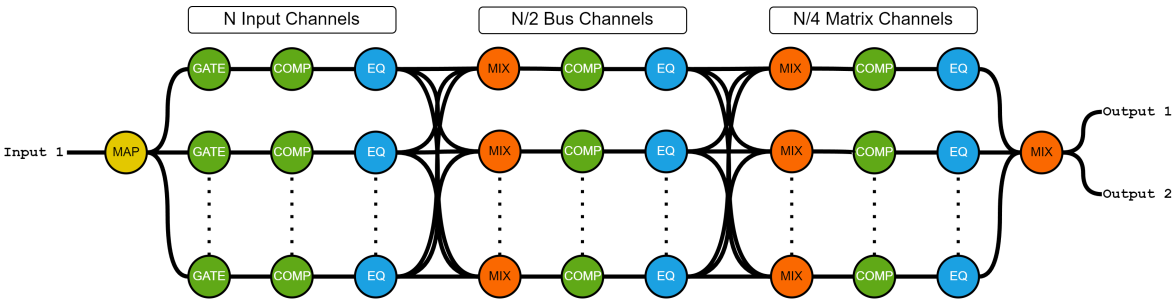


Figure 3.6.: Node structure of the signal graph mimicking a multi-stage mixing console.

3.13.2. Mixing Console Comparison

To test whether the Jetson can function as a mixing console, I built a signal graph that mimics the three-stage signal flow of a mixing console. The signal graph is shown in Figure 3.6.

The input stage consists of N parallel input channels. Each channel is processed using a noise gate, a compressor, and a five-band equalizer.
The input channels are then mixed down into $N/2$ bus channels and another compressor and five-band equalizer audio effect are used for each bus channel.
The bus channels are mixed down into $N/4$ matrix channels, each containing the same audio effects as the buss channels.

A few details do not precisely match the signal flow of a mixing console.
Firstly, there are restrictions concerning the parallelism of host $\leftrightarrow$ device copy operations. As a result, copying a large number of input channels, each in individual buffer segments is inefficient. This could be solved by merging the segments and copying them using a single command. However, neither the test setup nor the signal graph API were designed to handle this. Additionally, compared to the execution time of the entire signal graph, the time difference between copying a one-channel buffer and a multi-channel buffer is negligible. Therefore, all lanes in the input stage get their input from the signal graph's single input channel. The same reasoning applies to the output stage, where the matrix channels are mixed down into a single stereo output channel.
Secondly, all compressor instances are replaced with a second noise gate since I did not have time to implement a compressor. Fortunately, these two audio effects are very similar and the substitution not skew the results at all.
Lastly, the signal graph does not support side-chain inputs or effect sends, which adds significant complexity to a mixing console.
However, all these limitations can be addressed in future work.

Furthermore, the Jetson itself is at a disadvantage in this experiment.
The test ignores channel routing flexibility, a significant advantage of the CUDA graph structure. The Jetson can redistribute channels between the stages or even add or remove entire stages at will.
Additionally, the signal graph includes $2N + \frac{N}{2} + \frac{N}{4} = \frac{11}{4}N$ fully serial FxGate instances, and needs to process $3N \cdot 5$ serial biquad IIR filters. As a result, the performance of such a signal graph will significantly increase, once these two audio effects are further optimised.

For these reasons, the results of this experiment should only be taken as an indication of Jetson's potential as a mixing console.

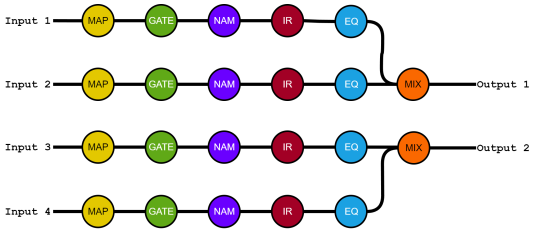
The execution times of the signal graph have been tested for increasing numbers of parallel channels until the Jetson could not process them within the time constraint given by the period size. A channel configuration test was considered successful if 99% of the process cycles were completed within the period size. Without this softening of the success criteria, all tests would have failed due to the launch irregularities described in Section 3.4.
At period sizes of 32 frames, the highest successful channel configuration was $N = 24$. At period sizes of 128 frames, the highest successful channel configuration was $N = 64$.

The results of this experiment are presented in Section 5.8.1.

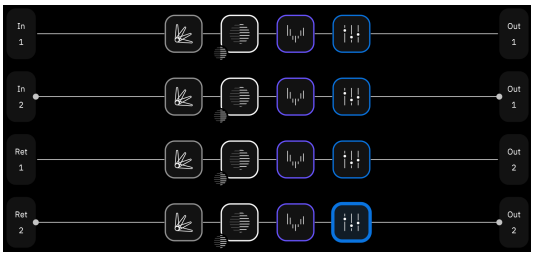
3.13.3. Quad Cortex Comparison

For the comparison against the Quad Cortex, I tested how many instances of similar audio effects each device could handle in parallel and series. Additionally, I compared the execution times of the signal graphs shown in Figure 3.7, which can be built identically on both devices.
Both signal graphs contain the same audio effects: a noise gate, an amplifier, a cabinet IR and an equalizer. The graph shown in Figure 3.7a focuses on maximizing the amount of parallel audio effects and is limited to four lanes due to the Quad Cortex's UI. The graph shown in Figure 3.7c focuses on complex routing and dynamic channel count handling.

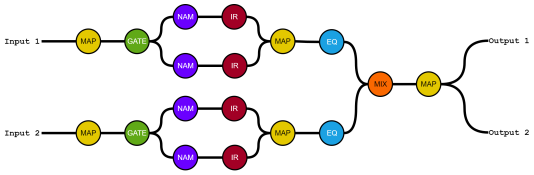
The results of these experiments are presented in Sections 5.8.2, 5.8.3 and 5.8.4.



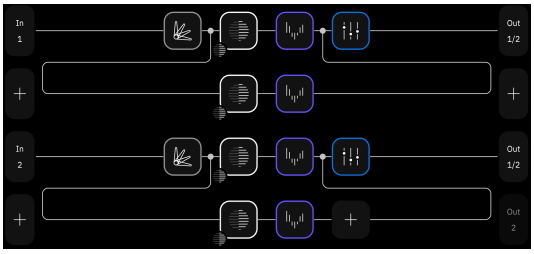
(a) Four Channel Stereo Guitar Signal Graph on the Jetson



(b) Four Channel Stereo Guitar Signal Graph on the Quad Cortex



(c) Two Channel Dual Amplifier Guitar Signal Graph on the Jetson



(d) Two Channel Dual Amplifier Guitar Signal Graph on the Quad Cortex

Figure 3.7.: Visualisation of the signal graphs used to compare the Jetson against the Quad Cortex.

4. Software Design

This chapter describes the final software design of the audio processing pipeline. The corresponding source code can be found on GitHub.

4.1. Overview

This section describes the software architecture and the control flow of the audio processing pipeline.

4.1.1. Object-Oriented Programming: Factory Pattern

I strongly avert the standard C++ practice of declaring classes in header files and defining their methods in source files, as this practice separates the declaration of private variables from their usage in the method definitions. To address this, I extensively use the Object-Oriented Programming (OOP) factory pattern. This pattern enables the definition of interfaces in the header files using classes with pure virtual methods. The implementation is then defined in the source files using classes inherited from the abstract interface classes. This removes any protected or private code from the header files for pure interfaces. As a result, there are fewer occasions to edit the header file, which would lead to the recompilation of all files that include it. Unfortunately, this pattern does not work as well for class inheritance between libraries.
It is discouraged to include body files from other libraries. For this reason, any protected code that should be available for deriving classes in other libraries should be located in the header file. The `Evaluator` class described in Section 4.6.2 is an excellent example.

4.1.2. Architecture

The code is distributed over multiple libraries to isolate external dependencies as much as possible and reduce compile times. A visual representation of the library dependencies and their most important classes is shown in Figure 4.1.

The **`utils`** library contains utility functions and classes that are used by all the other libraries As a result, this library should not link any of the other libraries to avoid circular dependencies.

The **`utils_cuda`** library contains the utility functions and classes that depend on CUDA. These include the CUDA error handling function macro, encoding and decoding of PCM signals and CUDA type operator definitions. While this library depends on the **`utils`** library, it should not link any other libraries to avoid circular dependencies.

The **`audio`** library contains audio-related code intended for execution on the CPU. Currently, all audio processing is performed on the GPU, and this library only contains the drivers, audio file IO, and interfaces for the audio processing classes. However, with mapped memory available, outsourcing highly serial algorithms like IIR filters to the CPU should be considered for future work.

The **`audio_cuda`** library contains the audio processing classes that run on the GPU. This includes the implementations of the signal graph and the audio effect interfaces declared in the **`audio`** library.

The **`eval`** library contains code designed to evaluate the accuracy and performance of audio processing classes.

The `trt` library contains the classes required to read an ONNX model, build a TensorRT engine and run inference on the GPU. This code is separated from the `audio_cuda` library mainly to reduce the compile time when tweaking the audio effect implementations.

The audio libraries are designed as an API that a UI can use to configure signal graphs. Unfortunately, due to time constraints, implementing such a UI had to be postponed for future work. For now, signal graphs must be configured and executed using the main file.

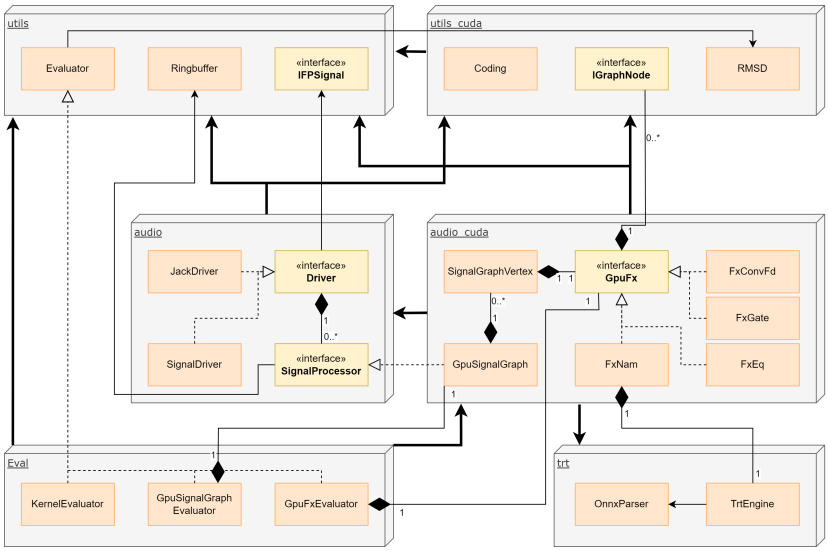


Figure 4.1.: Library dependency overview

4.1.3. Control Flow

The sequence diagram in Figure 4.2 illustrates the control flow of setting up and running the audio processing pipeline. The response arrows and some less relevant participants have been omitted to simplify the diagram.

Configuration

First, the UI instantiates one of the audio driver implementations. The `JackDriver` is used in this example because it is currently the only real-time driver implementation available. Next, the user has to select the audio effects that should be applied to the audio signal and build them into a signal graph. The signal graph is then registered with the audio driver, along with lists of inputs and outputs it should process and write to.

Setup

Once started, the audio driver sets up all members of the audio processing pipeline. In the case of the `JackDriver`, this includes:

- Setting the buffer size and registering the process callback with the Jack backend
- Registering the configured input and output ports with the Jack backend
- Setting the buffer size, input and output channels on the signal graph

The signal graph uses the provided channel information to configure its audio effects. Some audio effects have fixed channel setups, while others, like the `FxEq`, can accommodate different channel configurations within specific limits. Therefore, the audio effects configuration is not finalized until the signal graph determines the number of input and output channels. In addition to the channel configuration, the period size is used to calculate and allocate the necessary memory for each audio effect.

Processing

Once all components are set up, the audio driver first starts the signal graph's processing thread to let it warm up before starting the Jack client. The processing pipeline now operates independently of the main thread. The green elements marked in the diagram represent the three main steps of the processing loop, which run asynchronously in no defined order. The inner loop of the signal graph thread is executed synchronously with the the outer loop. Its green colored block is not meant to signify another asynchronous section but a limitation of the sequence diagram tool.

The Driver Thread The Driver Thread is responsible for periodically providing input data to and consuming output data from the signal graph. In the case of the `JackDriver`, the interval is determined by the process callback of the Jack audio server.

Update Fx Params Since the UI created all audio effects, it still has pointers to all of them. As a result, it can change the non-destructive parameters of the audio effects at any time by calling the update method.

Signal Graph Thread The signal graph thread is responsible for feeding the driver's input data into the audio processing pipeline and returning the same amount of processed samples for playback. Using a set of ring buffers at this point allows decoupling between the period count and size of the driver and the signal graph. For example, USB 1.0 audio interfaces should be run using multiples of 48 sample buffer sizes, as described in Section 3.2. The signal graph, however, prefers buffer sizes that are multiples of 32 to maximize the utilisation of the CUDA warp scheduling. Additionally, the size of the ring buffers can be adjusted to increase the signal graph's acceptable Worst-Case Execution Time (WCET). It would be preferable to have only a single ring buffer per path instead of one in the Jack audio server and one in the signal graph. However, since the Jack API offers limited control over its ring buffer, additional buffering is required to hide the CUDA launch irregularities.
Because ring buffers are used, the pointers to the source and destination buffers for the CUDA process graph change for each processing cycle. Therefore, before launching, the signal graph needs to update the CUDA process graph with the current I/O pointers. After this, both block buffers are advanced. As a final step, the update method is called on each audio effect with the instance of the CUDA process graph, allowing them to update their graph nodes with changed parameters.

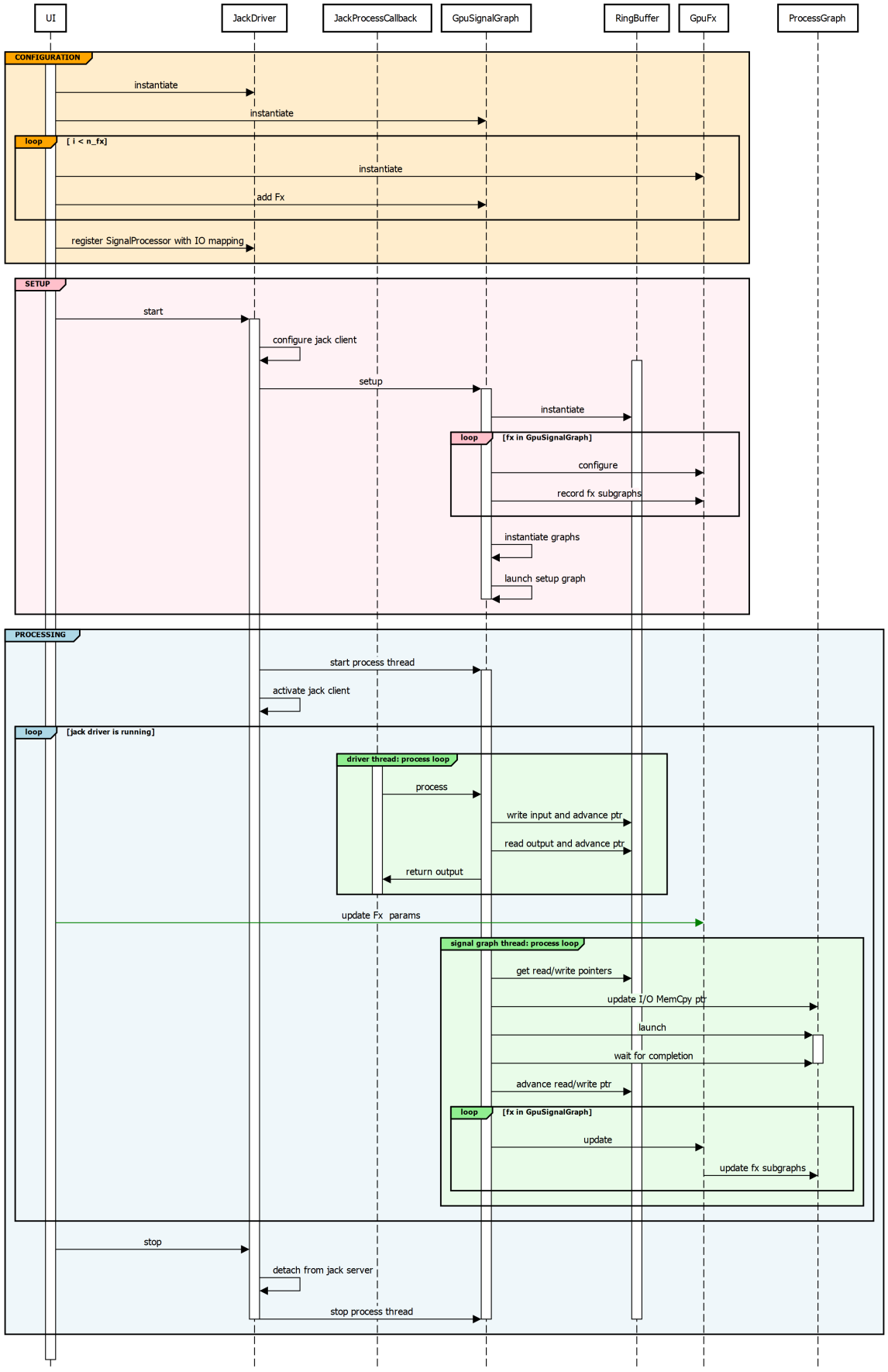


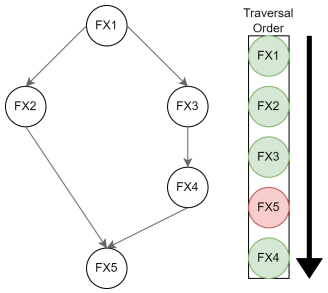
Figure 4.2.: Control flow of the audio processing pipeline

4.2. Audio Driver

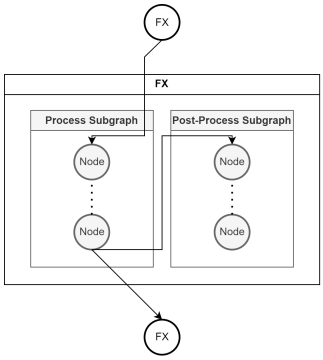
The audio driver interface is designed to be simple, exposing only methods for starting/stopping the driver and for getting/setting parameters such as buffer size, sample rate, and bit depth. Any interface implementation must include the start and stop methods, but it is not required to include the getters and setters. For example, the Jack C++ API does not provide methods to update the sample rate or bit depth.
The start method is intended to handle all the driver and signal graph setup work, while the stop method should handle all the teardown work. This ensures that the UI can stop and restart the driver without any memory leaks or other side effects. It is important to note that changing the driver's buffer size requires reallocating of all the buffers used by the signal graph. Since this is a time-consuming operation, the setter method should stop the processing loop, update the buffer size, and then restart the processing loop.

JackDriver

Most of the code that would comprise an audio driver implementation is handled by the Jack C++ API. This leaves the JackDriver to register callbacks and I/O ports, set the buffer size and activate the processing loop.
To avoid duplicate channel processing, I/O port names configured by the UI are stored in a set. After registering the ports with the Jack backend, the returned pointers are stored in a map with the port names as keys. At the start of each process cycle, the driver retrieves the buffers for each port and forwards them to the signal graph.
Aside from the process callback, the JackDriver can also register xrun and shutdown callbacks. These would be useful for implementing dynamic signal graph period count adjustment and graceful shutdown behaviour in the future.



(a) Visualisation of how a normal breadth-first search traverses the fifth vertex before the fourth and therefore violates the dependency order.



(b) Visualisation of how the process and post-process CUDA subgraphs of an audio effect are added to the CUDA process graph of the signal graph.

Figure 4.3.: Signal graph structure examples.

4.3. Signal Graph

The signal graph has two main responsibilities:

- manage the dependencies between the audio effects
- build update and run the CUDA setup and process graphs

It combines audio effects into a directed acyclic dependency graph consisting of signal vertices (named as such to avoid confusion with the CUDA graph nodes). Each signal vertex contains a single audio effect or memory copy command, a list of its parent and child vertices and pointers to the input and output buffers required by the audio effect.

The setup process of the signal graph is split into three steps:

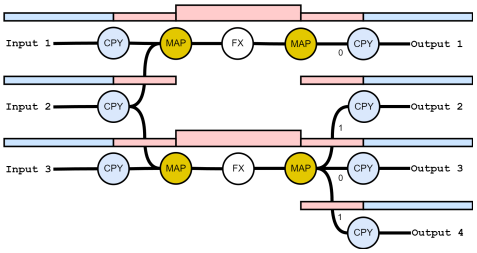
Input Mapping At the start of a process cycle, the signal graph takes a list of pointers to segmented channel buffers as input. To process these buffers with audio effects, they must be copied into device memory and interleaved. Instead of having each root vertex directly reference the host input buffers, the signal graph adds 1D memory copy vertices for each input channel buffer as new roots to the graph. This prevents scenarios like the one shown in Figure 4.4b where the expensive host-to-device memory copy is performed twice for the `input 2`.
As depicted in Figure 4.4a, signal graphs with multiple input channels must start with an FxInputMap audio effect. With it, the UI can restrict a lane of the graph to process only a subset of the input channels. If the signal graph has only one input channel as presented in Figure 4.4c, there is no need for an extra interleaving step.
There are signal graph configurations where it would be more efficient to interleave the inputs using the host to device copy commands. However, the CUDA graph API does not allow developers to change the pointers of 2D memory copy nodes in an instantiated graph. Under the `memset` and the `memcpy` graph update restrictions, the CUDA runtime API documentation states that "only 1D memsets nodes can be changed".[12] If Nvidia meant to include memcpy operations with this statement, they should change the wording to make it more clear.
Nevertheless, using 2D memory copy nodes as roots or leaves for the CUDA graph is impossible.

Vertex Traversal With the input buffer pointers sorted out, the graph's vertices are traversed in an adapted breadth-first search. The problem with a simple breath-first search is that it does not guarantee that all parents of a node are traversed before the node itself,as shown in Figure 4.3a. Since the CUDA graph node of a signal vertex requires the CUDA graph nodes of its parent to specify its dependencies to the CUDA graph, it may not be traversed before its parents. Therefore, the algorithm returns vertices to the queue if it has unvisited parents.

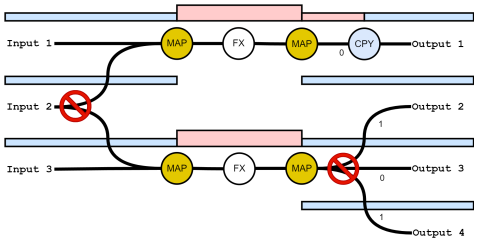
Each vertex performs the following steps:

- If the audio effect can not be configured to consume the combined input channel count of all parents, an exception is thrown
- If it has multiple parents, a single parent with multiple children or differing I/O channel counts, in-place processing is not possible, and a new destination buffer is allocated
- Build the audio effect's setup, process and post-process subgraphs
- Add the setup subgraph to the setup graph with no dependencies to ensure all setup subgraphs are executed concurrently
- Add the process subgraph to the process graph with dependencies to the process subgraphs of its parents
- Add the post-process subgraph to the process graph with a dependency to the current process subgraph

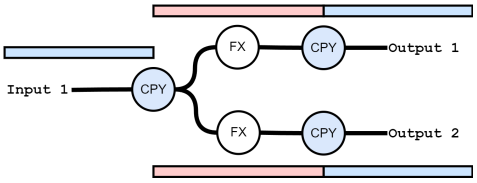
Output Mapping The signal graph leaves should contain instances of the FxOutputMap audio effect, which the UI can use to route the signal graph outputs to multiple output channels, as shown in Figure 4.4a. This step can be skipped if the signal graph has only one output channel as shown in Figure 4.4c, since deinterleaving is unnecessary.
Similar to the input mapping stage, the signal graph adds 1D memory copy vertices for each channel buffer as new leaves to the graph to handle the device to host memory copy.



(a) Complex I/O routing where multiple input and output mappings are required.



(b) Example of a duplicate host to device memory copy due to a missing input vertex and the more efficient 2D memory copy operation at the output, whose buffer pointers unfortunately can not be updated between graph executions.



(c) Simple I/O routing where not mappings are required.

Figure 4.4.: Visualisation of different I/O routing use cases. The blue and red rectangles represent the buffers required to pass audio between the audio effects. Blue buffers are allocated in the host and red buffers in the device context. The horizontal length of the rectangles does not represent the buffer size, but indicates which audio effects share the same buffer. The vertical length of the rectangles indicates the number of channels the buffer contains.

4.3.1. Post-Process Subgraph Execution

Initially I had planned to create a separate CUDA post-process graph and run it after every CUDA process graph execution. However, since the post-process tasks of the currently implemented audio effects are simple, they are better placed as leaves in the CUDA process graph. Since no subsequent nodes depend on the CUDA post-process subgraphs, they will not block the execution of any following subgraphs.
Still they will be executed concurrently with whatever resources are unused by the process subgraphs. Admittedly, this will delay the availability of the processed output. However, if the process graph's execution time runs against the period duration, any work done after the process graph may delay the execution of the following process graph. Conversely, if the process graph execution time is far below the period duration, the added delay does not matter.

4.3.2. Buffer Allocation Strategy

By definition, all audio effects must support in-place processing, meaning that their source and destination pointers may point to the same memory location. This constraint allows serially dependent audio effects with identical input and output channel configurations to work on a single buffer. The signal graph must only allocate separate buffers if the graph flow splits, merges or channel counts change. This behaviour is illustrated in Figure 4.5.
At the start, both input channels are copied to the device, each requiring an individual device buffer. The `FxInputMap` audio effects then restrict the two lanes to one channel each. Due to their ability to have differing input and output channel counts, the input mapping nodes allocate separate output buffers by default, even though that is unnecessary in this example.
The `FxGate` audio effect can only process a single channel and, therefore, works in place.
Next, the `FxNam` audio effects split the lanes. Therefore, each requires separate output buffers.
The following `FxConvFd1c1` instances are single-channel audio effects and, therefore, process in-place.
Another set of input mapping nodes interleave the mono outputs of the four lanes into two stereo lanes.
The `FxEq` audio effect can process multiple channels as long as the input and output channel counts are the same. As a result it can process in-place on the stereo buffer.
The `FxMixSegmented` audio effect mixes the stereo lanes into a single stereo output.
Finally, the `FxOutputMap` audio effects deinterleaves the stereo output into two mono buffers to be copied back to the host context.

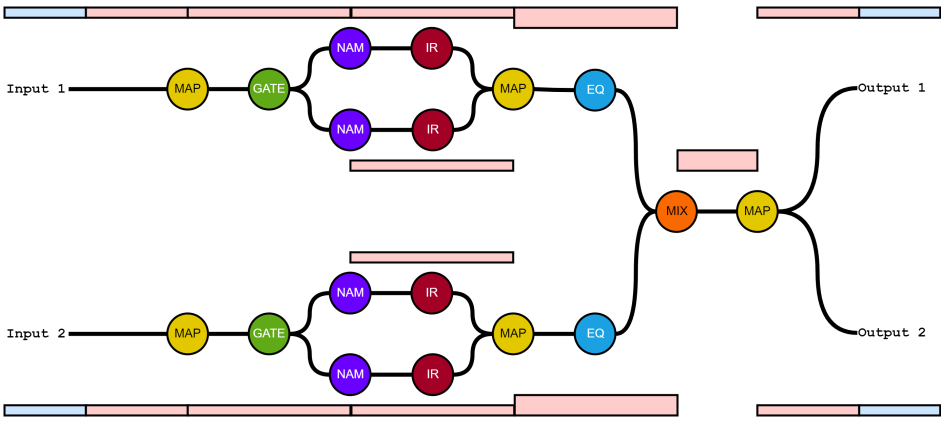


Figure 4.5.: Visualisation of the buffer allocation strategy for a complex signal graph. Each rectangle represents a distinct buffer allocated on the host (blue) or device (red) context. The length of the rectangles specifies how many audio effects share the same buffer. The height of the buffer specifies the number of channels it contains.

Name	Description	Input Channels	Output Channels
FxConvFd2c2	Convolves a stereo input with a stereo IR	2	2
FxConvFd2c1	Convolves a stereo input with a mono IR	2	2
FxConvFd1c1	Convolves a mono input with a mono IR	1	1
FxNam	Distorts a mono input signal in the style of the amplifier, the neural network was trained on	1	1
FxEq	Equalizes n channels using one or more biquad filters	n	n
FxGate	Mutes a single channel if its amplitude falls below a given threshold	1	1
FxInputMap	Interleaves n `BufferSegments` into a single `BufferSegment`	n	1
FxOutputMap	Deinterleaves an interleaved `BufferSegment` into n `BufferSegments`	1	n
FxMixInterleaved	Mixes an n-channel interleaved `BufferSegment` down to a m-channel interleaved `BufferSegment`, given $n \mod m = 0$	n	m
FxMixSegmented	Mixes $l = n/m$ lanes of n/l-channel `BufferSegments` downto a single m-channel interleaved `BufferSegment`, given $n \mod m = 0$	n	m

Table 4.1.: Overview of the implemented audio effects

4.4. Audio Effects

All audio effects are derived from the abstract class `IGpuFx`. This interface defines all the public methods the signal graph requires to uniformly execute the three control flow phases described in Section 4.1.3 on all audio effects. The `setup`, `process` and `postProcess` methods can either be called directly for stream launching, or through the `recordSetupGraph`, `recordProcessGraph` and `recordPostProcessGraph` methods for graph recording.
Input and output data of all audio effects is uniformly handled using a collection of `BufferSegment` pointers contained in a custom `Buffer` class. Each audio effect defines its source and destination buffer requirements using the `BufferSpecs` class and, during processing, reads or writes data to and from the `Buffer` accordingly.

The Table 4.1 provides an overview of the implemented audio effects.

4.4.1. FxConvFd2c2

The `FxConvFd2c2` audio effect is used to convolve a stereo signal with a stereo impulse response.

To create an instance of the audio effect, the user has to provide an impulse response and a maximum FFT buffer size. Impulse responses representing room reverberation are often very long to capture the entire reverb tail. The maximum FFT size is used to clip impulse responses that are too long for real-time processing. The actual FFT buffer size is then set to the next higher power of two to take advantage of the cuFFT library's performance improvements.

During the setup of the audio effect, all the internal buffers are allocated and set to zero where necessary. The interleaved stereo impulse response is copied to its prepared complex buffer and processed using a single C2C forward transform. The resulting complex buffer is then unpacked into two separate complex buffers representing the left and right channels of the impulse response. Refer to Section 3.3 for more information on how and why this process works.

During each process cycle

- the source buffer is copied into its FFT-sized dry buffer
- the dry buffer is transformed using a C2C forward pass and unpacked the same way as the impulse response during the setup phase
- the left and right dry buffers are multiplied with the corresponding complex IR channels (only half of the buffers are processed due to the FFT symmetry)
- both convolved buffers are transformed back to the time domain using individual Complex-to-Real (C2R) inverse transforms
- the left and right mono buffers are repacked into an interleaved stereo buffer
- the convolved signal is summed with the residual buffer, and the result is written to the wet buffer
- using the mix ratio parameter, the dry and wet buffers are mixed and written to the destination buffer

Finally, the post-process phase rewinds the residual buffer by the process buffer size and clears buffers where necessary.

FxConvFd2c1

This frequency-domain convolution audio effect variation is used to convolve a stereo signal with a mono impulse response. While Room Impulse Responses (RIRs) are often stereo, guitar cabinet IRs are almost always mono. Additionally, after convolving a stereo signal with a mono IR, the transformation back to the time domain can be done in a single inverse C2C pass, removing multiple copy and an entire FFT operation from the process. The only downside is, that the IR has to be copied to both the real and imaginary channels of the complex IR buffer. Fortunately, this is a one-time operation during the setup phase.

FxConvFd1c1

This frequency-domain convolution audio effect variation is used to convolve a mono signal with a mono impulse response. It uses the standard Real-to-Complex (R2C) and C2R FFT transformations for the convolution process and is only faster than the other variations due to the lower channel count.

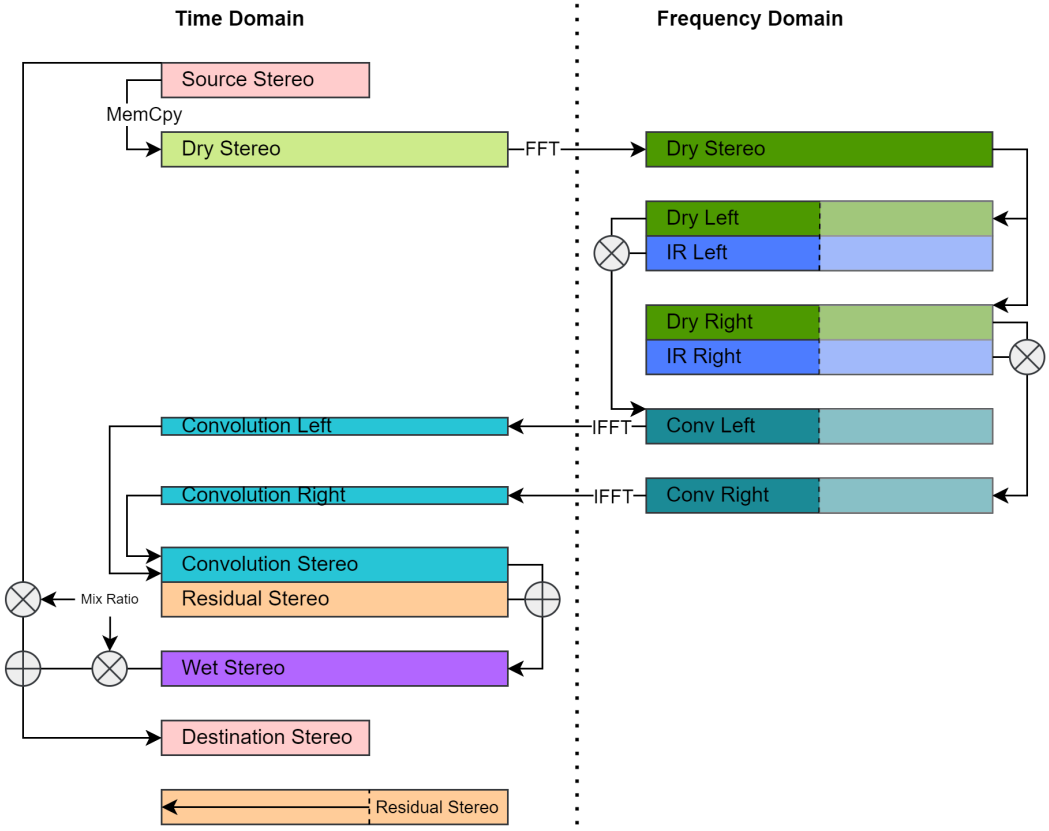


Figure 4.6.: Dual Stereo Guitar Processing Signal Graph

4.4.2. FxNam

The `FxNam` audio effect uses the TensorRT library to run inference on a pre-trained ONNX export of the convolutional neural network designed by Steve Atkinson. The network takes a mono signal as input and distorts it to sound as similar as possible to the sound of a guitar amplifier on which it was trained. For this purpose, the input is run through two serial stacks of dilated 1D convolutional layers with dilations in powers of two from 1 to 2048 per stack. This results in a receptive field of $R = 2 \cdot 2^{12} = 8192$.[21] Translated into audio terms, this means that the prediction of each new sample depends on the previous 8192 samples ($170.667ms$).
In real-time audio processing, the period size will always be smaller than 8192 samples. As a result, the audio effect has to maintain a history buffer of the last R samples to feed into the network. The history buffer is extended by the period size p to ensure that the length of the output matches the length of the input.

To create an instance of the audio effect, the user has to provide the path to the ONNX model and desired TensorRT engine precision.

During the setup of the audio effect, the TensorRT engine is built from the ONNX model for the configured precision and process buffer size. This process is quite time-consuming, so the generated engines are stored in a file to reduce the setup time for subsequent sessions. Afterwards, the history buffer is allocated to the length $L = R - 1 + p$ and set to zero.

During each process cycle, the source buffer is copied to the rightmost area of the history buffer and fed into the TensorRT engine. Since the size of the output tensor is $p = L - R - 1$, it is directly written to the destination buffer.

Finally, the post-process phase rewinds the history buffer by p samples to prepare it for the next process cycle.

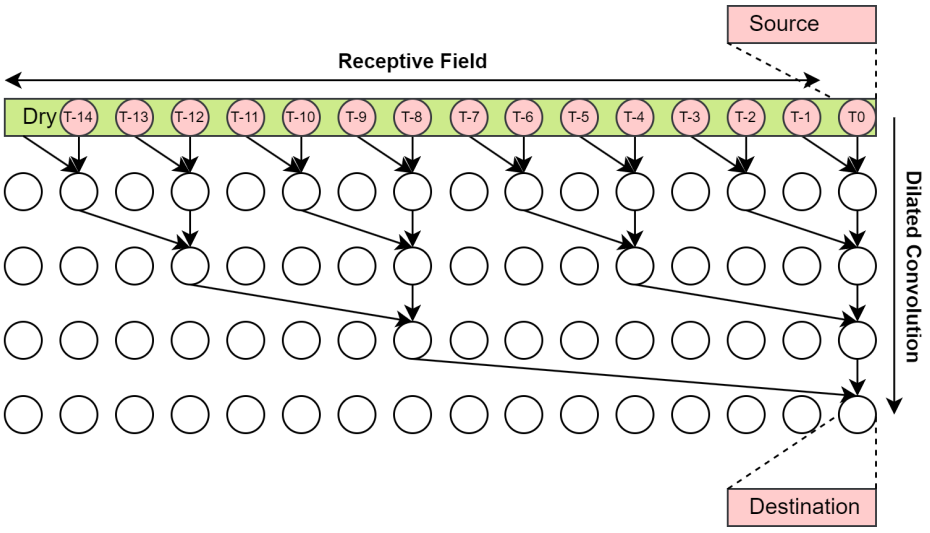


Figure 4.7.: Simplified visualisation of how dilated convolutional layersincrease the receptive field of a neural network. Note that the dilation works on sample granularity and not the buffer granularity shown in the image.

4.4.3. FxEq

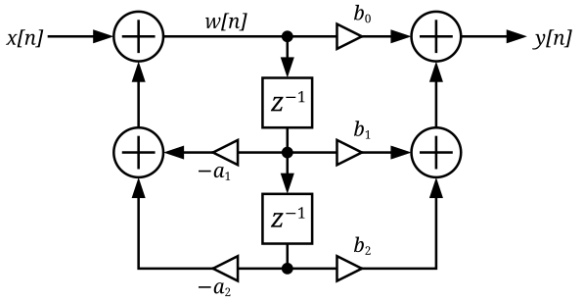
The audio effect FxEq uses the biquad filter formula to apply EQ curves to the input signal. A biquad filter comprises a second-order feed-forward (FIR) and a second-order feedback (IIR) network. The audio effect implements the Direct Form 2 shown in Figure 4.8a to half the number of history buffers and remove the need for memory copy commands between the EQ bands.

To create an instance of the audio effect, the user has to provide the biquad filter parameters for each required band. The BiquadParam classes for peak, low-/high-shelf and low-/high-pass filters use the formulae from the Audio EQ Cookbook by Robert Bristow-Johnson[25] to calculate the biquad filter coefficients from gain, frequency and Q parameters.

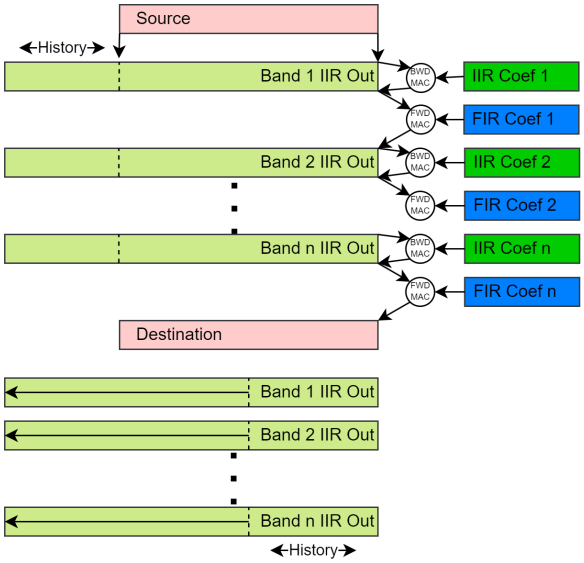
During the setup of the audio effect, the processing buffers for all EQ bands are allocated. All of these buffers are larger than the period size by a number of frames equal to the filter degree, enabling them to hold the history of the last period. Furthermore, the filter coefficients are calculated for each band and copied into device memory.

During each process cycle, the source buffer is copied to the rightmost area of the process buffer of the first EQ band. Due to the recursive nature of the IIR network, the IIR kernel is well suited to an in-place operation, as each result has to be written back to the same buffer for use in calculating the next result. It is launched with a single thread to prevent out-of-order execution. The FIR kernel has no such restrictions and is launched with the optimal launch arguments for a period-sized kernel determined in Section 3.10. The execution order of the Direct Form 2 implementation allows the IIR kernel to process in-place on the process buffer of its EQ band and the FIR kernel to use out-of-place processing to write the result to the next EQ band buffer.
From there, the process is repeated for each EQ band in series, and the output of the last EQ band is copied to the destination buffer.

Finally, the post-process phase rewinds the process buffers by the period size for all EQ bands.



(a) Direct Form 2 of a Biquad Filter[24]



(b) Visualisation of the FxEq audio effect

4.4.4. FxGate

Last but not least in a list audio effects that are essential to the guitar signal chain is the noise gate. Note that this audio effect has intentionally been implemented without parallelism to show that even the most serial audio effects run well on a GPU.
The purpose of a noise gate is to reduce a signal's noise floor by muting it when it falls below a certain threshold. To prevent artefacts from abrupt changes in volume, the gate applies a smooth gain transition using exponential decay curves based on attack and release time parameters. Hysteresis can be prevented using either a hold time parameter or a separate threshold for closing the gate. This implementation uses the former approach.

To create an instance of the audio effect, the user has to provide a dynamics parameter object containing all the parameters required for the gate.

During the setup of the audio effect, there is no need to allocate any processing buffers for once. Instead, device pointers for the state variables like the current gate state, hold counter and gain are allocated and initialized to the default values. Lastly, the attack and release time parameters are converted to exponential decay factors.

During each process cycle the gate kernel is launched with a single thread, and the source and destination buffers as arguments. The kernel then iterates serially through all the samples of the current buffer and performs the following steps depending on the current gate state:

- `Closed`: set the gain to zero and change the state to `Release` if the current sample is above the threshold
- `Release`: increase the gain using the inverse release decay factor and change the state to `Open` once the gain approaches one
- `Open`: set the gain to one and change the state to `Hold` if the current sample is below the threshold
- `Hold`: increment the hold counter and change the state to `Attack` once the counter reaches zero
- `Attack`: update the gain using the attack decay factor and change the state to `Closed` once the gain approaches zero

At the end of the loop, the current sample is multiplied with the gain and written to the destination buffer.

This audio effect does not require a post-process phase.

4.5. CUDA Graph Node Wrappers

The CUDA graph node wrapper classes are designed to simplify adding a command as a graph node during stream capturing using the hybrid method described in Section 2.2.2. As a side benefit, if the stream is not in capture mode, the command is launched into it instead. The `MemCpyNode`, `MemCpy2DNode`, and `KernelNode` are each able to handle a single CUDA node.

More complex use cases like interleaving and deinterleaving buffers require multiple CUDA nodes. After realising, that the `cudaMemCpy2D` commands for these two operations only differ in pitch and buffer type, I could not resist the urge to create a class hierarchy to handle them uniformly.
The `MultiMemCpyNode` class is an abstract base class that defines methods to retrieve pitch and buffer pointers from segmented and interleaved buffers. The `MemCpyInterleaved2SegmentedNode` then inherits from the `MultiMemCpyNode` and overrides the source pitch and pointer retrieval methods for an interleaved buffer and the destination pitch and pointer retrieval methods for a segmented buffer. The `MemCpySegmented2InterleavedNode` does the opposite. At instantiation, the `MultiMemCpyNode` class creates the required number of `MemCpy2DNode` using the overridden methods to retrieve the pitch and buffer pointers.

If any doubts were left about how much I enjoy OOP, this implementation should put the final nail into that coffin.

The Section A.2 shows the difference in lines of code between the hybrid graph building method and the custom node wrapper method for a mono to stereo interleaving operation.

4.6. Evaluation

4.6.1. Root Mean Square Deviation

The RMSD calculator is used to create reproducible test cases for the audio effects. It takes two buffers, the number of samples and the maximum offset as inputs. The memory order of the buffers is irrelevant as long as it is the same for both. The buffers can be of different sizes as long as the number of samples corresponds to the smaller buffer's size. The calculator launches $n*2+1$ RMSD calculations in parallel, where n is the maximum offset. Once all calculations are complete, the offset that produced the smallest RMSD and its RMSD value are returned. Since the buffer offset is computed in both directions, the returned offset is negative for a left shift of the expected buffer and positive for a right shift. In other words, if the returned offset is positive the algorithm under test has delayed the signal by that amount of samples.

4.6.2. Evaluator

The `Evaluator` is an abstract class that can be used to measure execution time statistics and test audio processing algorithms against an expected output. For this purpose, it defines the `measurePerformance` and `testAccuracy` methods that use the protected pure virtual methods `setup`, `test`, `process`, `postProcess` and `teardown` to prepare and then execute the algorithm under test.
The `measurePerformance` parameters include the number of warmup and measurement runs, the number of frames in the process buffer and whether the process cycle should be repeated back to back or with a simulated period duration delay. The option to run warmup cycles is required to avoid measuring the first few process cycles, which often show a sharp decrease in processing time before stabilizing. However, it is recommended to run some measurements without warmup cycles since the startup behaviour can vary between algorithm implementations and may need to be considered when comparing them.
The `testAccuracy` method expects two buffers containing the test input and the expected output. It then executes the algorithm under test using the `test` method and compares the output with the expected buffer using the RMSD calculator.
This setup enables the derivation of any number of specific evaluators to test certain parts of the audio processing pipeline without having to duplicate the performance and accuracy evaluation code. Each child class only has to implement the `setup`, `test`, `process`, `postProcess` and `teardown` methods to be able to use the `measurePerformance` and `testAccuracy` methods.
Examples of this practice can be found in the `eval` library.

5. Results

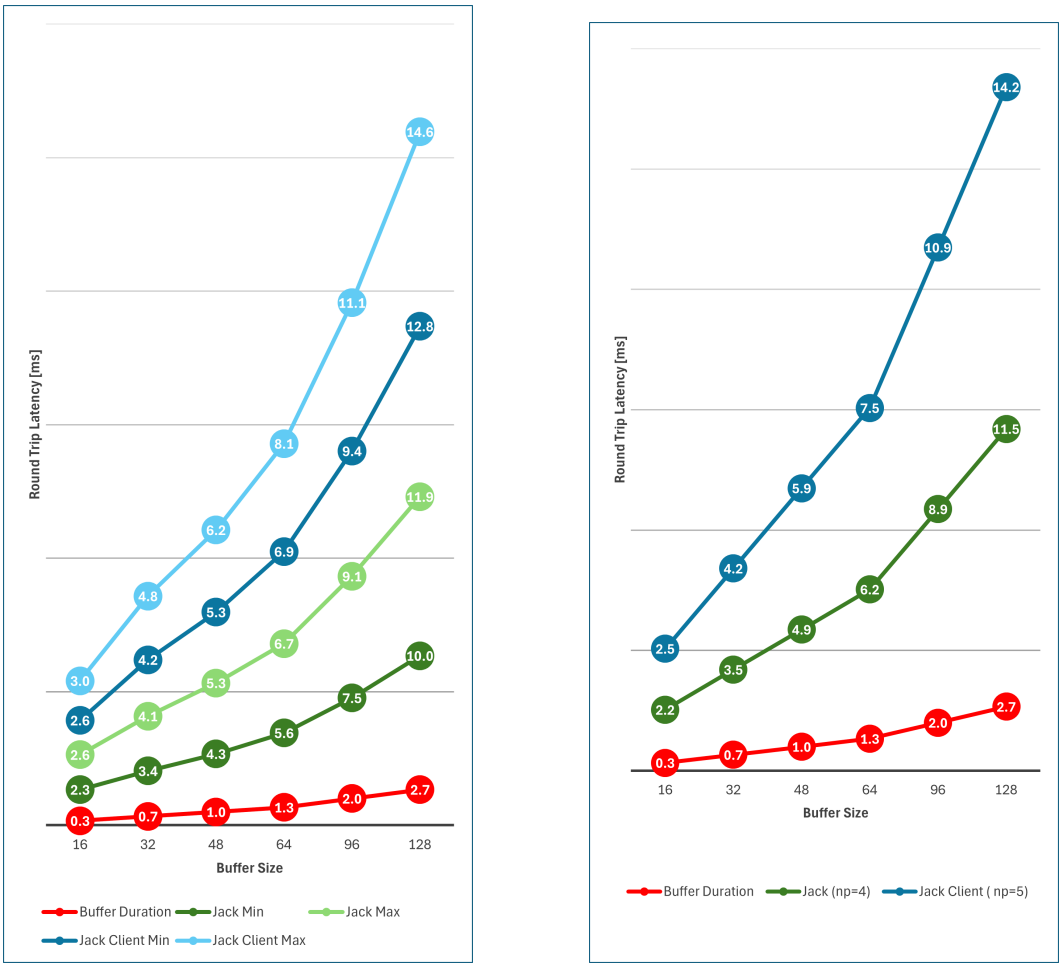
The following sections present the results of the experiments described in chapter 3.
Where not otherwise stated, execution time statistics were gathered using 1000 warm-up and 10000 measurement iterations. At lower numbers of warm-up iterations, subsequent measurements of the same experiment would show significant variance, rendering the results unreliable. This is attributed to the CUDA scheduler's diminished performance after having been idle for too long, as observed in Section 3.12.

5.1. Loopback Measurements

The Figure 5.1a shows the **Jack** and **Jack Client Loopback** measurements taken using the setup described in Section 3.1.4. The measurements were performed at 48 kHz sample rate, 24 bit depth, and for periods $p \in \{16, 32, 48, 64, 96, 128\}$ frames. The 16-frame period represents the lowest possible configuration of the Focusrite Scarlett 8i6 audio interface. The 128-frame period represents the largest acceptable buffer size for real-time audio processing as defined in Section 3.1.1. The Direct Monitoring latency $t_{DM} = 0.62ms$ is not affected by the buffer size and is therefore not included in the plot.

Readers may be surprised by the existence of minimum and maximum values for **Jack** and **Jack Client Loopback** configurations. Repeated latency measurements of the same settings have shown fluctuations ranging from $\sim 0.3ms$ at $p = 16$ up to $\sim 1.8ms$ at $p = 128$. A similar behavior was observed when performing the same measurements on the Behringer Uphoria UM2 audio interface. Crossvalidating the results using the `jack_delay` tool by Fons Adriaensen removed the measurement setup as a potential cause for the variance. Therefore, Jack or the USB driver must have caused the fluctuations.

As described in Section 2.2.4, starting the audio server with $n = 2$ periods, Jack adds $n_p = 4$ periods to the round-trip latency. One USB frame is enough to transmit one period of audio data. Therefore, the USB 2.0 protocol will at least add $2 \cdot t_{USB} = 0.25ms$ to the round-trip latency. Lastly, the Direct Monitoring latency is used to approximate the latency t_H introduced by the audio interface hardware. With these constants, I was able to approximate the expected round-trip latencies as a function of the period size $f(p) = t_H + 2 \cdot t_{USB} + n_p \cdot \frac{p}{f_s}$.
With $n_p = 4$, the approximations closely match the **Jack** loopback measurements. The measurements for the **Jack Client Loopback** tend to be approximately one period higher than the **Jack Loopback** measurements. Jack seems to require one additional latency period to interface with other C++ code. This conclusion is supported by the fact that the `jack_delay` tool, implemented in C++, reports measurements matching the approximations with five latency periods.
The number of latency periods added by Jack and the measurements' inconsistencies reinforce the need for a better audio interface and custom audio driver to elevate this proof of concept to a prototype. For example, the Quad Cortex produces no significant latency fluctuations and has round-trip latencies of $\sim 1.5ms$ and $\sim 1.9ms$ for measurements comparable to the **Jack** and **Jack Client Loopback** signal paths, respectively. For now, these measurements can be used to abstract and validate the measurements of the audio processing pipeline.



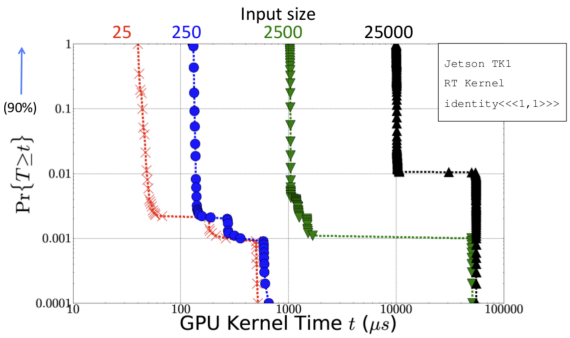
(a) Measured Loopback Latencies

(b) Approximated Loopback Latencies

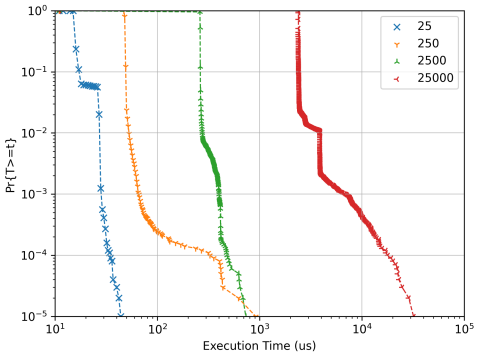
Figure 5.1.: Comparison of measured and approximated loopback latencies for the Focusrite Scarlett 8i6 audio interface using Jack. Note that the plot uses stacked series and the scale should be ignored in favor of the data labels.

5.2. Kernel Launch Irregularities

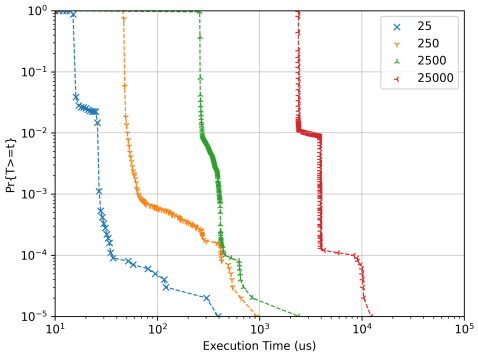
Figure 5.2 compares kernel launch irregularities between measurements taken from a 2017 study[1] and the measurements taken in this thesis. The measurements in all three cases were performed on an identity kernel that copies the input buffer to the output buffer for single channel buffers of $f \in \{25, 250, 2500, 25000\}$ frames. The original goal was to only measure the launch time variability. Therefore the kernels were all launched with only a single thread.
The results show that the variability has not improved significantly in the last seven years. However, Jetson Orin NX used in this thesis processes the kernels significantly faster than the Jetson TK1 used in the 2017 study. This reduces the impact of the irregularities since they are less likely to exceed the real-time limit.
Unsurprisingly, the CUDA graph API does provide better stability than the stream API for single kernel launches. It requires more complex workloads to show its full potential, as demonstrated in Section 5.6.



(a) Identity kernel launch irregularities using the CUDA stream API measured in 2017 study.[1]



(b) Identity kernel launch irregularities using the CUDA stream API in this thesis.



(c) Identity kernel launch irregularities using the CUDA graph API in this thesis.

Figure 5.2.: Comparison of the launch irregularities of an identity kernel using the CUDA stream API and the CUDA graph API.

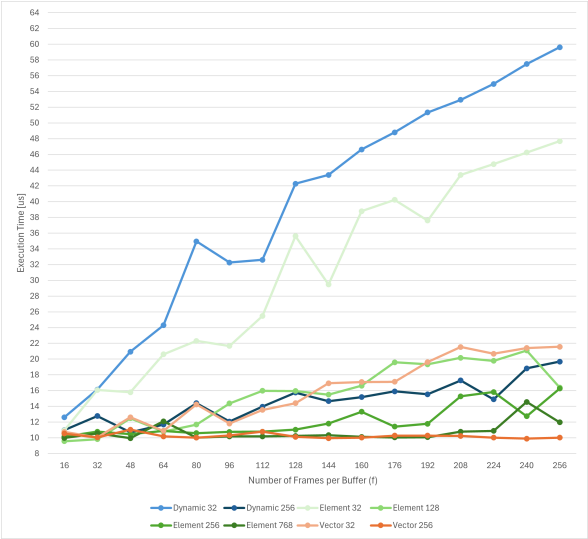
5.3. Vector Memory Instructions

Figure 5.3 presents the measurements of the experiment described in Section 3.9. All kernels were launched with a single block, as increasing the block count did not significantly improve performance at the buffer sizes tested. The number next to the kernel name indicates the number of threads n_{tpb} the kernel was launched with. All measurements were taken for $c = 4$ channels, where a single frame of $4 \cdot 4$ byte `float` values fully utilises a vector load instruction of 16 bytes. Due to the extreme fluctuation of the maximum execution times shown in Figure 5.3c, the figures Figure 5.3b and Figure 5.3d are clipped $50us$ and $100us$ for the $n_{tpb} = 32$ and $n_{tpb} = 256$ measurements respectively.
The **Dynamic** kernel performed significantly worse than the other two kernels. This indicates that the CUDA scheduler cannot infer the memory access pattern of the nested loop at launch time. This problem could be solved using loop unrolling. However, this was not deemed necessary due to the **Element** and **Vector** kernel results.

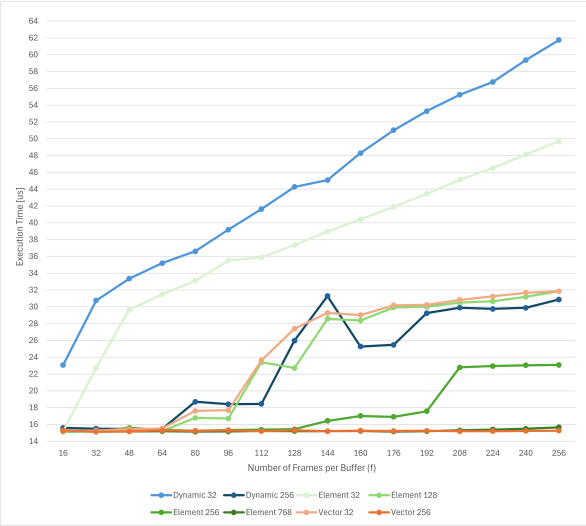
The **Element** and **Vector** kernels performed equally well when comparing their measurements for fairly matched thread counts. At thread counts **128** and **32** both show an apparent increase in execution time as the frame count further exceeds the thread count. At thread counts **768** and **256** the execution time remains largely constant. At identical thread counts, the **Element** kernel performs worse than the **Vector** kernel, although only once the frame count exceeds half the number of threads.

According to the CUDA Programming Guide, memory transactions are coalesced for entire warps.[10] As a result, the memory instruction overhead for both implementations would be identical as long as the entire memory area required by the warp resides in a contiguous, interleaved buffer. Furthermore, whether accessing 4 `float` or a single `float4` value, both align perfectly with the maximum 16-byte memory transaction size. Therefore, the execution time of these two kernels should primarily depend on the number of launched threads, which is reflected in the measurements.

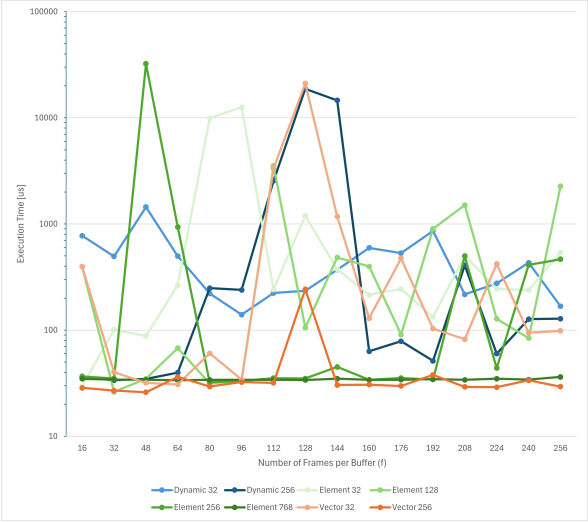
From these observations, **Element** kernels require at least $n_{tpb} = 2 \cdot f \cdot c$ threads, and **Vector** kernels require at least $n_{tpb} = f \cdot c$ threads for optimal performance. The only downside to using **Element** kernels is the increased thread count required to achieve optimal performance. Therefore, I recommend using **Element** kernels at the start of development to reduce the amount of code that needs to be maintained. In later stages, the **Element** kernels can be replaced by multiple fixed-channel **Vector** kernels to reduce the required thread count of established audio effects. For fixed-channel audio effects, I recommend using **Vector** channels from the start, as removing channel striding from a thread loop produces much cleaner code.



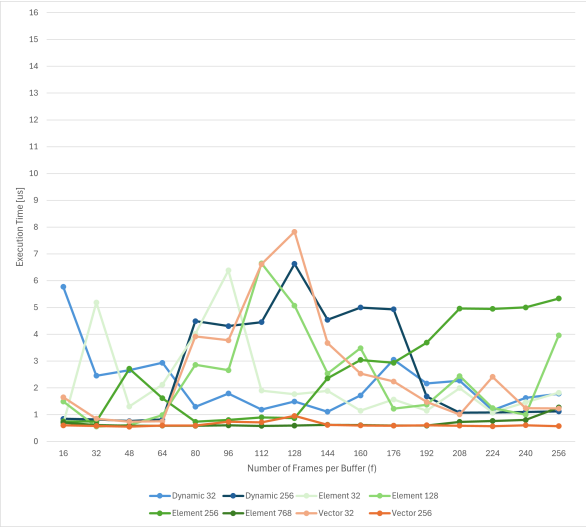
(a) Vector Load Min



(b) Vector Load Clipped Mean



(c) Vector Load Max (logarithmic scale)



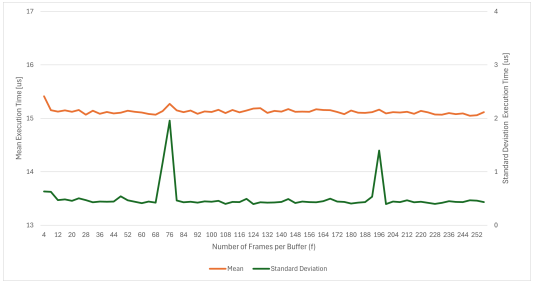
(d) Vector Load Standard Clipped Deviation

Figure 5.3.: Maximum, mean and standard deviation of the execution times of three kernels with differing vector load instructions. The measurements are color coded with blue for the **Dynamic** kernel, green for the **Element** kernel and orange for the **Vector** kernel. Lighter shades represent lower thread counts.

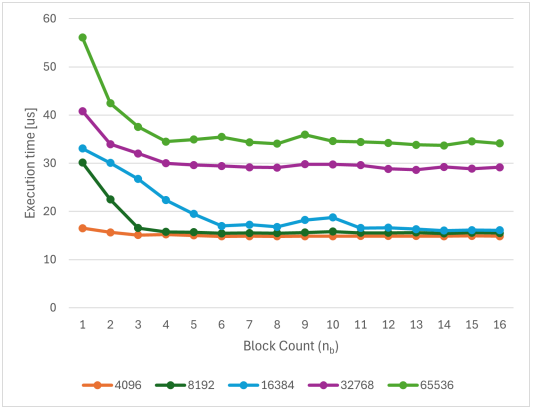
5.4. Kernel Launch Arguments

Figure 5.4a shows the mean and standard deviation of the execution times of the `FxEq` FIR kernel for different buffer sizes f at a channel count $c = 1$. The kernel launch arguments for each experiment were set to the optimal values of $n_b = 1$ and $n_{tpb} = 2 \cdot f \cdot c$ for an element kernel as determined in Section 5.3.
The measurements show virtually no fluctuation between different buffer sizes. The spikes in the standard deviation series at $f \in \{76, 196\}$ are artefacts caused by the CUDA scheduler and are not representative of any stability issues for the buffer sizes tested.
These results are indicative for all implemented kernels operating on similar buffer sizes. Consequently, the application should always use the maximum period size that can be tolerated under the given real-time requirements.

Figure 5.4b shows the mean execution times of the complex multiplication kernel used in the `FxConvFd2c2` audio effect. Previous measurements have indicated that maximizing the threads per block n_{tpb} resulted in the best performance for all block counts n_b. As a result, the kernel was launched with $n_{tpb} = 768$ for all experiments based on observations described in Section 3.10
The measurements show how increasing the block count reaches a point of diminishing return at approximately $n_b = 4$ for all tested period sizes. The slight increase in execution time for $n_b = 9$ suggests that the scheduler distributes each block on an unoccupied SM, of which the Jetson has eight. At nine blocks, the scheduler would need to schedule two blocks on the same SM, which could explain the consistent spike in execution time at all tested period sizes.
From these observations, the kernels running on FFT sized buffers should be launched with$n_b = 4$ for an optimal tradeoff between performance and resource utilisation.



(a) FxBiquad FIR kernel buffer size n_f grid search results.



(b) FxConvFd Complex Multiplication kernel block count n_b grid search results.

Figure 5.4.: Most significant results of the kernel launch argument optimisation experiments.

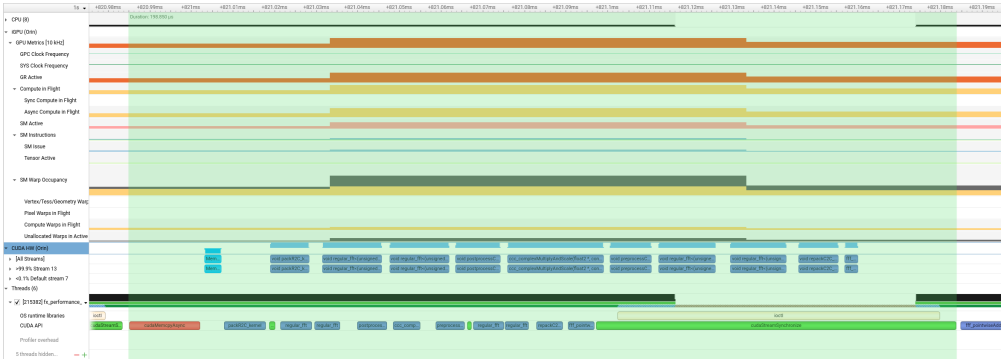
5.5. Convolution Optimisation

Figure 5.5 shows the different variations of the IR convolution audio effect profiled by the Nsight Systems software. The green-marked area encapsulates the execution of an entire process cycle of the respective audio effect. Note that the execution times shown at the top left are inflated due to Nsight Systems' profiling overhead and should only be used for comparison within this Section. Green blocks represent control flow commands like `cudaStreamSynchronize`, Red blocks represent memory operations like `cudaMemcpyAsync`, and Blue blocks represent kernel executions. All four audio effects were executed using the stream API.

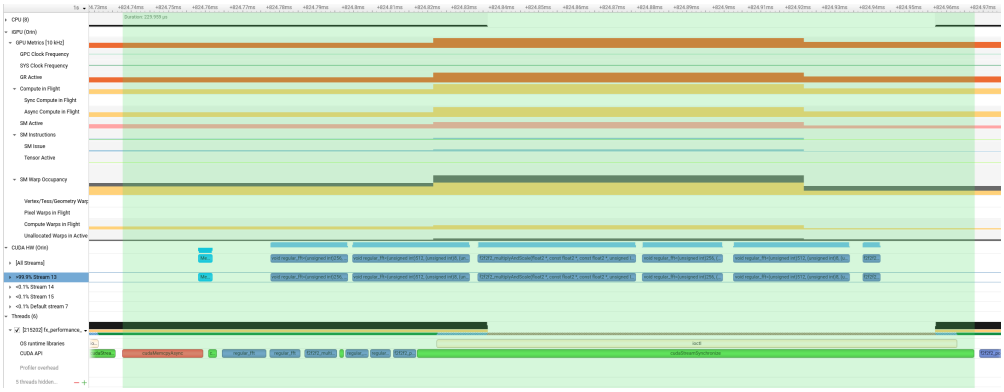
The profiles Figure 5.5a, Figure 5.5b and Figure 5.5c show how the kernels are launched into the stream as fast as possible. After approximately half of the execution time, the host's control flow has reached the end of the process cycle and is waiting for all work in the stream to be completed.

The profile in Figure 5.5a is a good example of why it is essential to avoid launching too many too-small kernels. Over half the process cycle is already spent on launching work into the stream. If this behaviour were to be exacerbated to the point where the kernels finish before the launch of the following kernel is completed, the stream would stall.

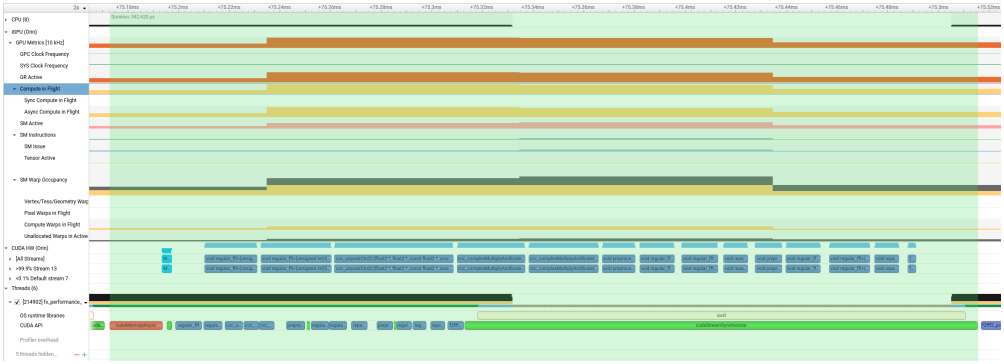
The profile in Figure 5.5d shows an attempt to further improve the performance of the FxConvFd2c2 audio effect using command level parallelism. The complex multiplication and the IFFT operation of the left and right channels can be performed in parallel and are, therefore, launched into separate streams.
Unfortunately, this requires the host to be synchronized at two additional points. First, to stream 13 before launching the complex multiply kernel of the right channel into stream 14. Second, to stream 14 before recombining the stereo channels into a single output buffer. While command-level parallelism saves time, the additional control flow complexity negatively offsets this benefit. As a result, this variation is slower than the original single-stream implementation.
It is likely that the parallel variation could be improved by using the CUDA Graph API. However, implementing this would have required a complete rebuild of the audio processing pipeline from hybrid stream capturing to strict graph building, which was not feasible within the scope of this thesis.



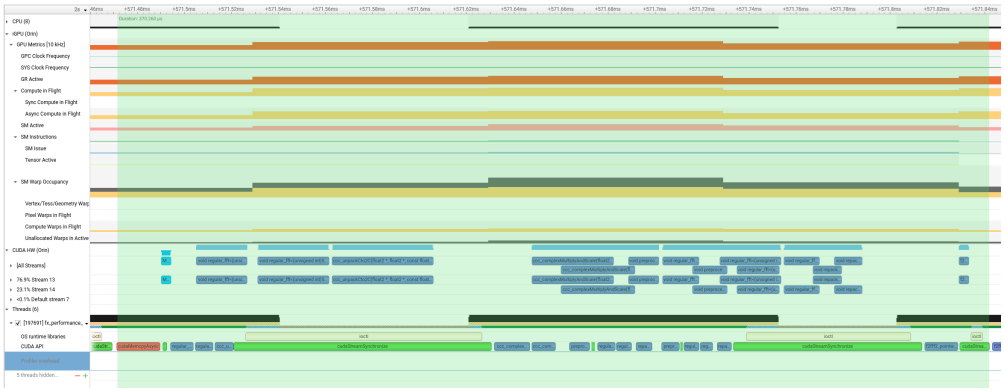
(a) `FxConvFd1c1` Mono impulse response convolution



(b) `FxConvFd2c1` Mono impulse response convolution with stereo input



(c) `FxConvFd2c2` Stereo impulse response convolution with stereo input



(d) `FxConvFd2c2` Stereo impulse response convolution with stereo input. All individual channel operations are parallelised on command level.

Figure 5.5.: Screen shots of the Nsight Systems profiling software comparing the execution of the IR convolution audio effect variations.

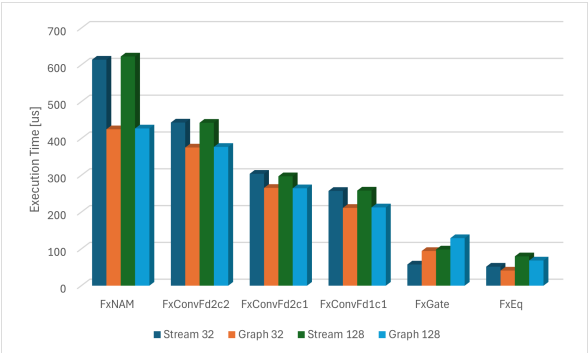
5.6. Stream vs Graph

The Figure 5.6 presents the execution times of all the audio effects described in Section 4.4 run using the stream and graph API for minimal and maximal period sizes. Additional performance-relevant arguments for specific audio effects are:

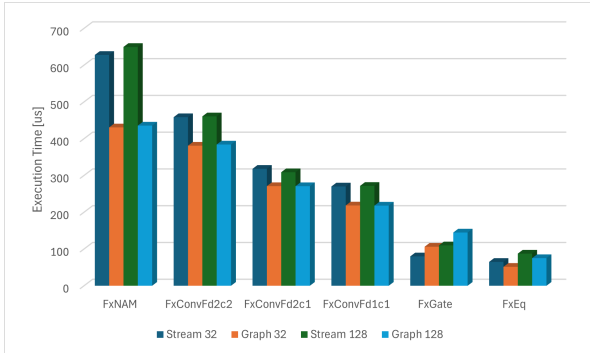
- The `FxConvFd` audio effects were setup with an FFT buffer size of $f = 2^{17} = 131072$ frames.
- The `FxEq` audio effect was set up with one EQ band

More complex audio effects, such as the `FxNam` and `FxConvFd2c2`, show a clear performance advantage when using the graph API for minimum and mean execution times. The difference between the audio effects reinforces the fact that the graph API is especially beneficial for audio effects combining many CUDA commands. The `FxNam` requires the most CUDA commands to calculate its 24 convolutional layers and, therefore, sees the most significant performance improvement.
Conversely, the `FxGate` audio effect launches only a single kernel with a single thread to force serial execution. As a result, it performs slightly worse when using the graph API.

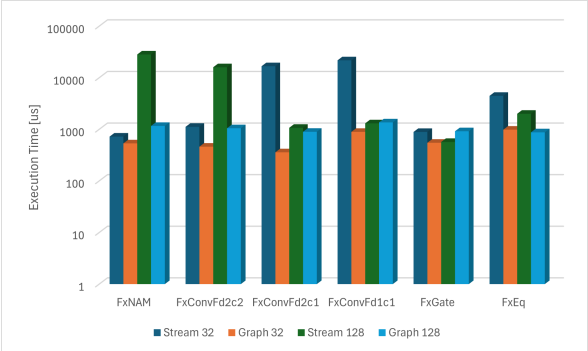
While the standard deviation and maximum execution times are erratic due to the CUDA launch irregularities, both show a clear tendency for graph execution to be more stable than stream execution. From these observations, the following results will only show the execution times of the graph API.



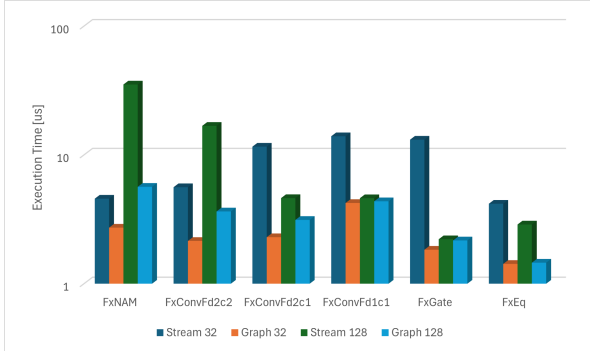
(a) Minimum



(b) Mean



(c) Maximum (logarithmic scale)

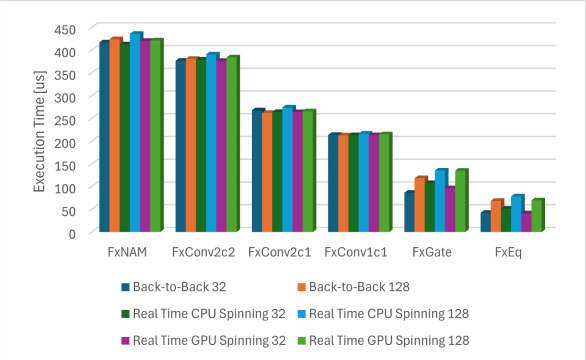


(d) Standard Deviation (logarithmic scale)

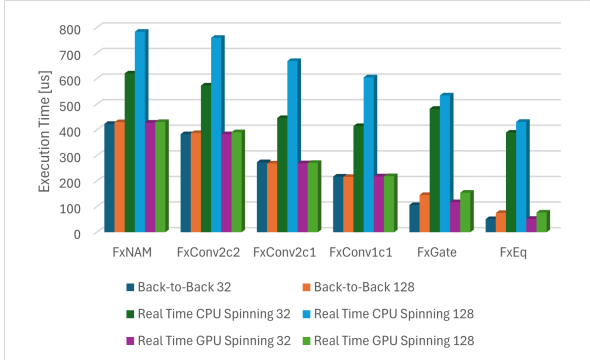
Figure 5.6.: Comparison of the execution times of all audio effects run using the stream and graph API for minimal and maximal buffer sizes.

5.7. Audio Effect Execution Time Irregularities

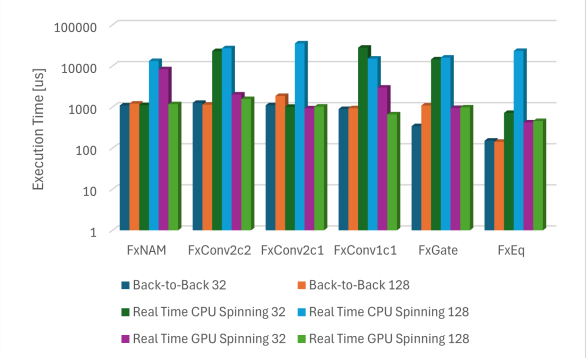
The Figure 5.7 presents the difference in execution times between back-to-back launches, real-time simulation with CPU spinning and real-time simulation with GPU spinning. The experiments were performed using identical audio effect settings as in Section 5.6.
Leaving the GPU idle during real-time period simulation increases the FxEq mean execution time by a factor of 10 and the standard deviation by a factor of 100. By switching from spinning on the CPU to spinning on the GPU, the statistics return to a close match to the back-to-back launch measurements. The fact that the minimum execution times are closely matched in all three scenarios indicate that this is a scheduling and not a processing issue. Even the maximum execution times show a clear tendency for CPU spinning to have the worst stability, which is only further emphasized by the standard deviation values.
I have no explanation for why the standard deviation for the `FxGate`, as the only completely serial audio effect, shows little improvement for GPU spinning compared to CPU spinning.



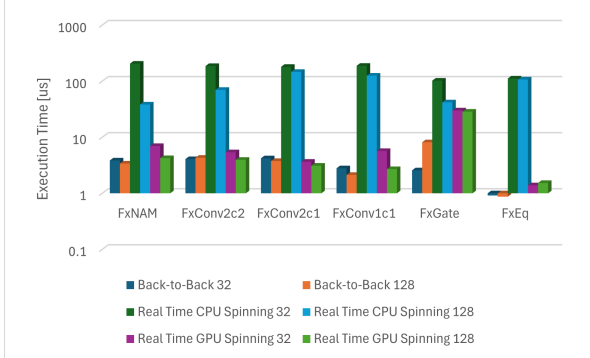
(a) Minimum



(b) Mean



(c) Maximum (logarithmic scale)



(d) Standard Deviation (logarithmic scale)

Figure 5.7.: Comparison of the execution times of all audio effects run using different inter-measurement-interval spinning methods.

5.8. Signal Graph Execution

This section presents the results of testing the Jetson's limits using the signal graphs described in Section 3.13. Each signal graph has been measured using both the **Signal Graph Processing Latency** and **System Round Trip Latency** signal paths.
The **System Round Trip Latency** for all signal graphs initially showed values at approximately one period higher than the **Jack Client Loopback Latency** measurements. However, these two latencies should be equal since the **Signal Graph Processing Latency** measurements were within the configured period duration.
The problem was that the output copy nodes of the signal graphs were added to the CUDA process graph without specifying their dependency towards the leaves of the audio effects graph. At the start of each process cycle, the output copy nodes transferred the output of the last audio effects before they were fully processed. As a result, the output had a constant delay of approximately one period in addition to the expected **System Round Trip Latency**. Fortunately, once found, the issue was easy to fix.

Since all signal graphs were confirmed to run in real-time at the expected **System Round Trip Latency** for the configured period size, the following results will only show the **Signal Graph Processing Latency** measurements.

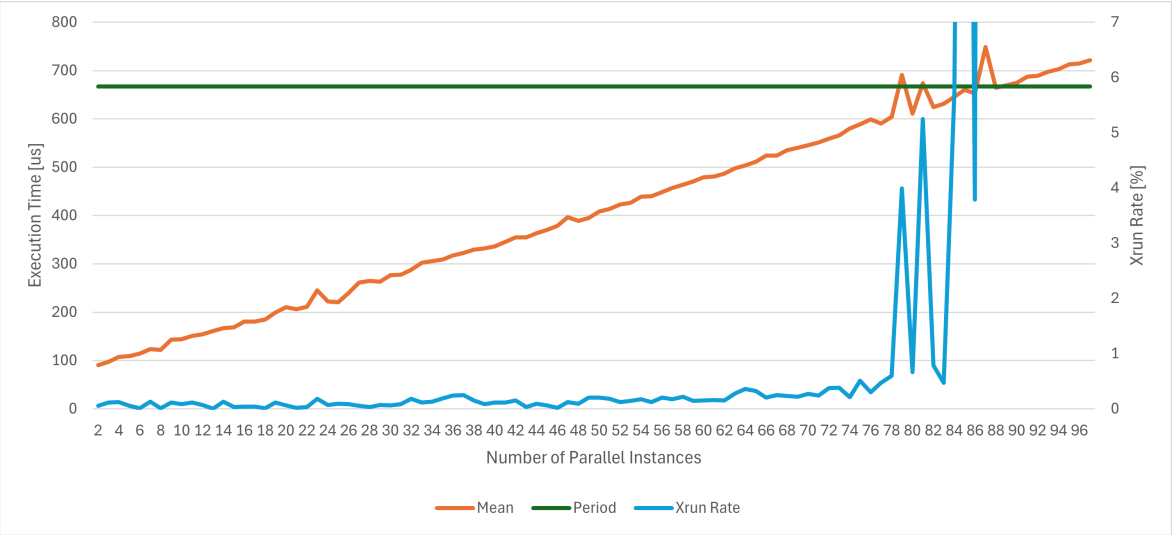
5.8.1. Mixing Console Signal Graph

Figure 5.8 tests the Jetsons limit in handling the channel routing of a three-stage mixing console at periods of 32 and 128 frames as described in Section 3.13.2. Each tested signal graph consists of N input channels, $N/2$ bus channels, and $N/4$ matrix channels, to a total of $1.75N$ channels.

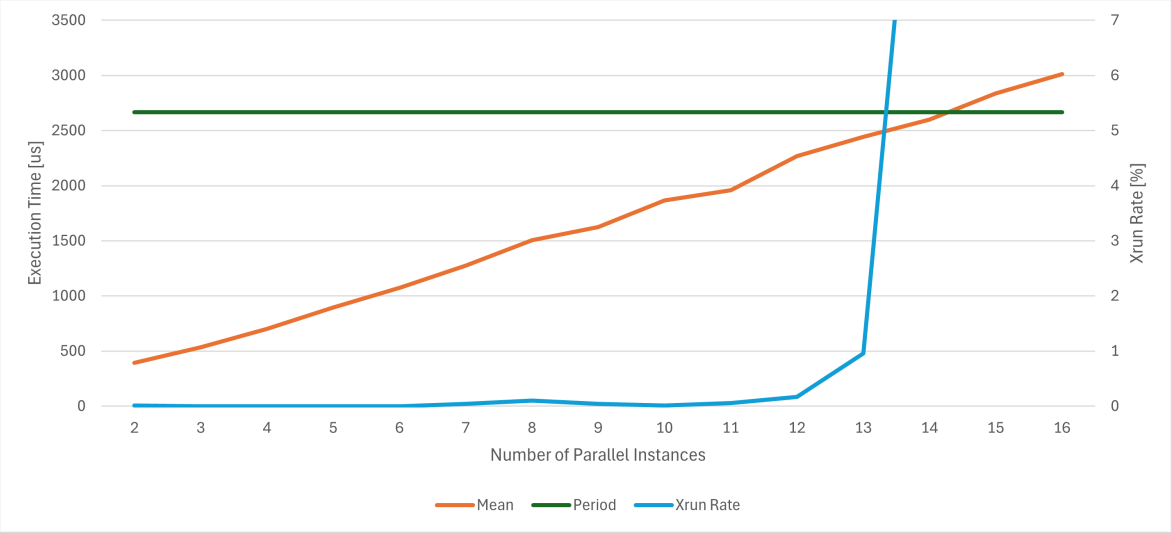
An experiment is considered successful if there were less than 1% xruns in the measurement iterations to account for the CUDA scheduler's launch irregularities. At a period of 32 frames, the Jetson can handle up to a $N = 24$ channel mixing console and at a period of 128 frames, it can handle up to a $N = 64$ channel mixing console.

The Nsight Systems profiles presented in Figure 5.9 show the execution order of the CUDA commands of a 48-channel three-stage mixing console style signal graph.
In the first iteration of the graph, the `FxMix` audio effect used a host-to-device copy command within the process cycle to make a list of pointers to the segmented input buffers available to the mixed kernel. These copy commands are depicted in Figure 5.9a as the small green blocks between the input, bus and matrix stages. Due to the API synchronisation behaviour described in Section 2.2.2, these copy commands act like a synchronisation barrier between the three stages, which results in a significant performance loss.

Figure 5.9b shows the same signal graph with the host-to-device copy commands moved outside of the process cycle. Suddenly, the input mix kernels of the bus stage are executed while other streams are still processing the post-process subgraph of the `FxEq` audio effect of the input stage. This is the exact behaviour that I hoped for when adding the subgraphs as leaves to the CUDA process graph as described in Section 4.3.1.

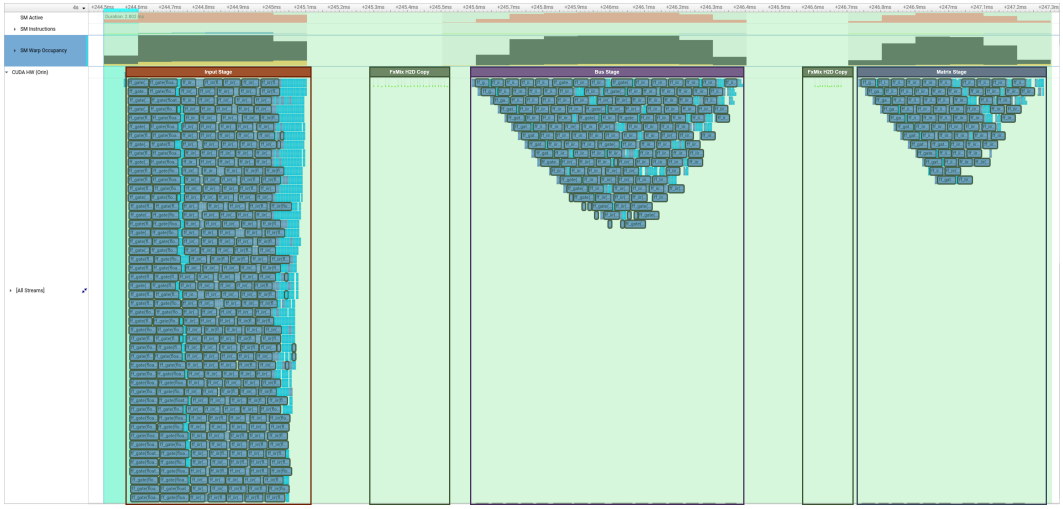


(a) Process Buffer Size: 32 Frames

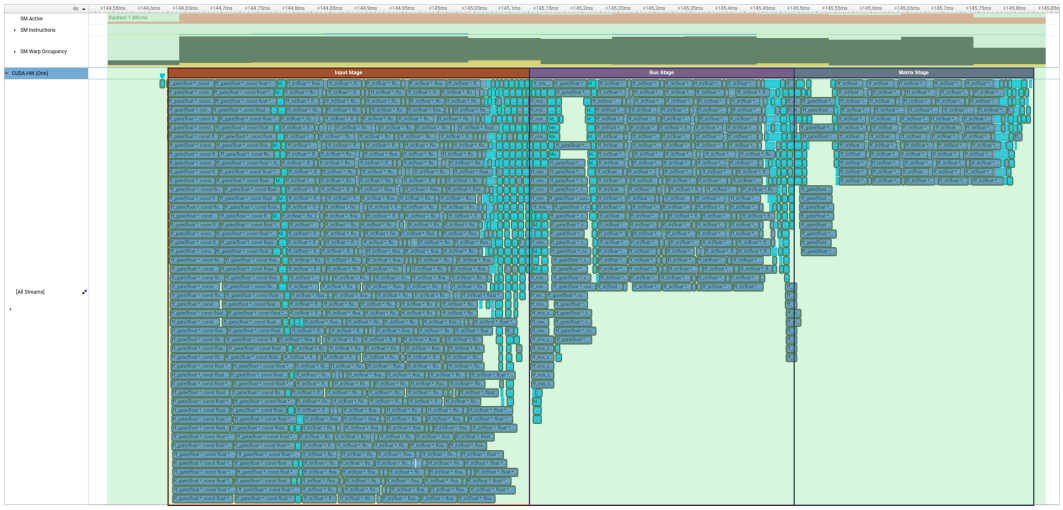


(b) Process Buffer Size: 128 Frames

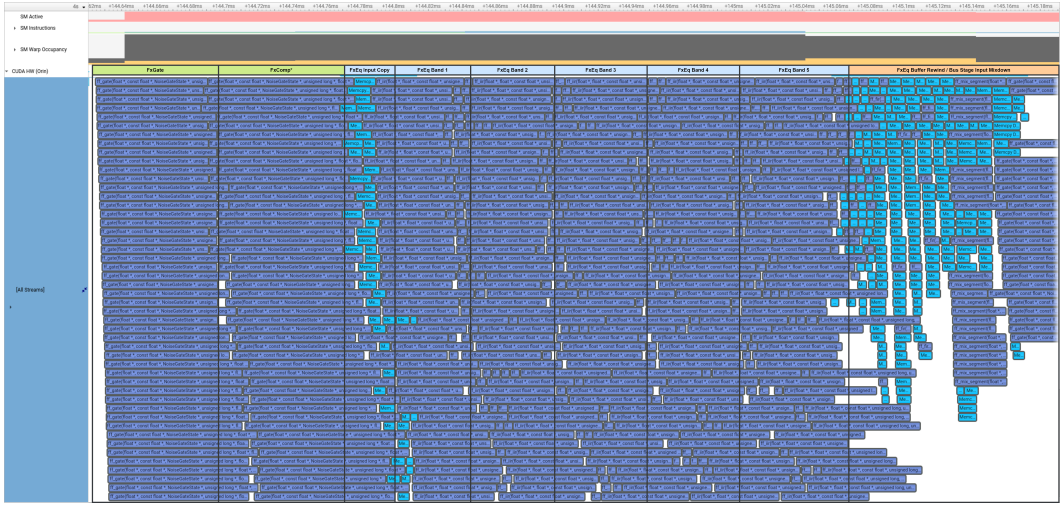
Figure 5.8.: Showcase of the Jetson's performance in mixing console style parallel channel processing. The number of xruns signifies how many process cycles missed the time constraint set by the period.



(a) Suboptimal signal graph execution with blocking host to device memory transfers.



(b) Optimal signal graph execution without blocking host to device memory transfers.

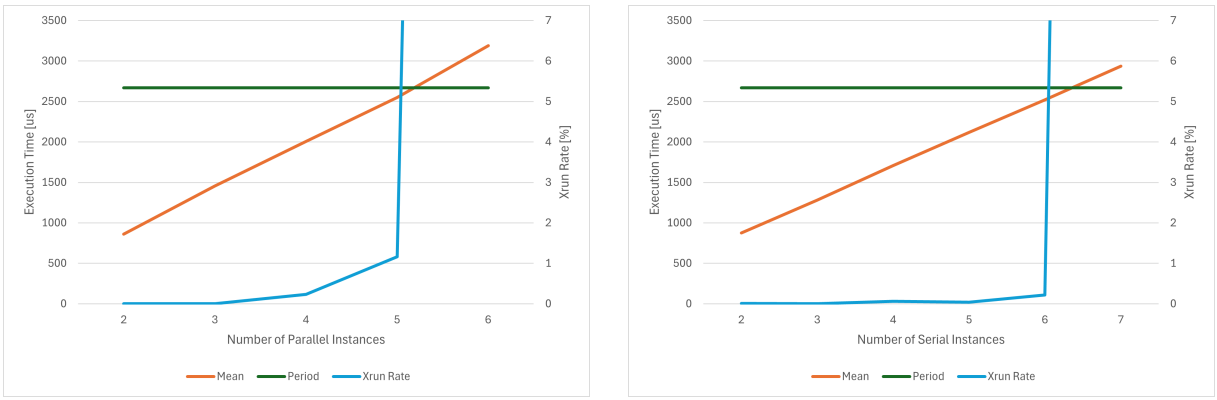


(c) Closeup of the input channel stage

Figure 5.9.: Screenshots of Nsight Systems profiles of a 48-channel three-stage mixing console style signal graph.

5.8.2. Neural Network Inference

Figure 5.10 presents how many instances of the Neural Amp Modeler ConvNets the Jetson can process in parallel and series using TensorRT. At a period of 32 frames, the Jetson can handle only one instance of the ConvNet. At a period of 128 frames, it can handle 6 instances in series and just misses the cutoff for the 5th instance in parallel.
These measurements and Nsight Systems profiles show, that each instance of the ConvNet can fully utilise the Jetson's GPU resources. As a result, there is no performance gain from running multiple instances in parallel.
The fact that the serial signal graph show better performance indicates that the TensorRT engine does not cooperate well with other workloads on the GPU.



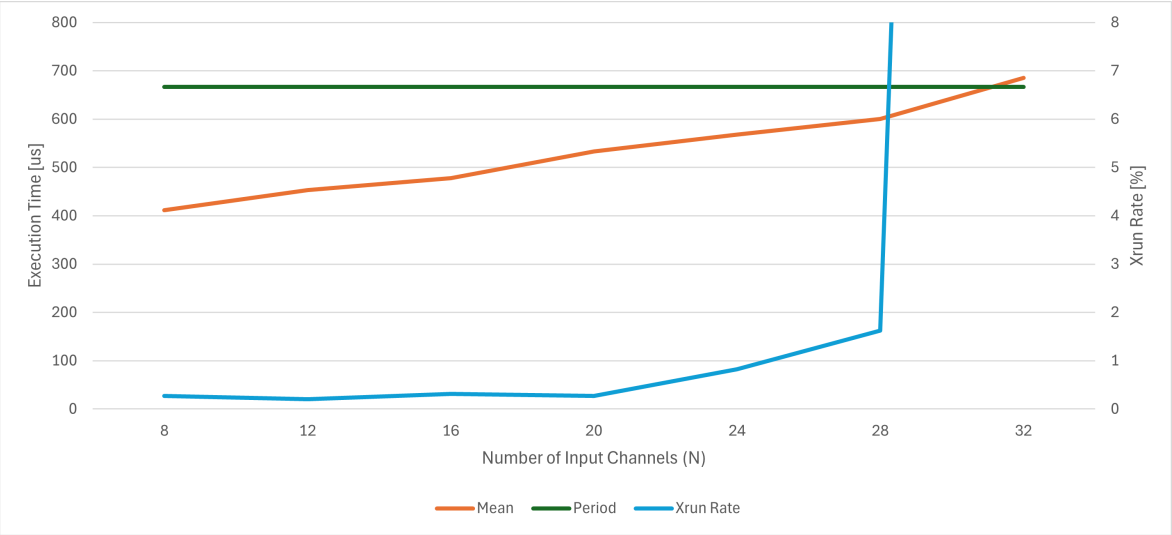
(a) Parallel (b) Serial

Figure 5.10.: Showcase of the Jetson's performance in neural network inference using TensorRT. The number of xruns signifies how many process cycles missed the time constraint set by the period of 128 frames.

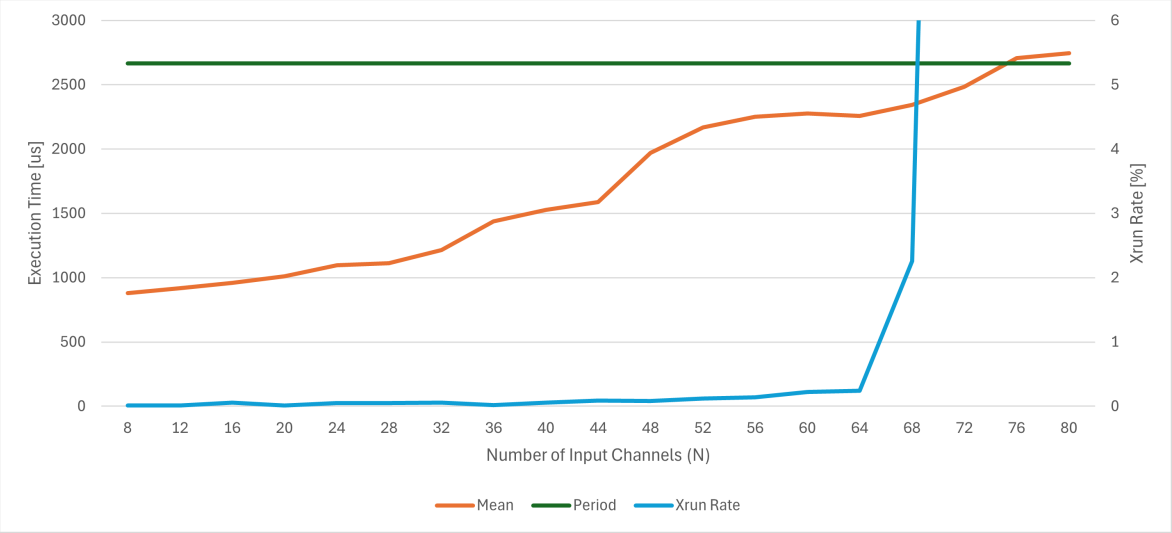
5.8.3. Parallel Convolutions

Figure 5.11a shows that the Jetson can run up to 78 parallel mono guitar cabinet IR convolutions of FFT size 4096 at a 32 frames period. This is eight times more than the Quad Cortex can handle. Since I do not know at what length the Quad Cortex processes the IRs, the comparison might not be entirely fair, but I doubt that it convolves with IRs much longer than $85ms$.

Figure 5.11b shows that the Jetson can run up to 13 stereo convolutions in parallel at a period of 128 frames. The impulse responses are clipped to 65'536 samples ($1.365s$).
Compared to the 1760 positional IRs the Vienna Power House extension to the Vienna MIR Pro 3D plugin by VSL can process on a consumer grade GPU, this number is not impressive. Unfortunately, I am again unaware of the exact specifications of convolutions performed by the Vienna Power House extension.



(a) Parallel `FxConvFd1c1` with 32 Frame Period and 4096 FFT Size (Guitar Cabinet IR Convolution)



(b) Parallel `FxConvFd2c2` with 128 Frame Period and 65'526 FFT Size (RIR Convolution)

Figure 5.11.: Showcase of the Jetson's performance in parallel mono and stereo convolution. The number of xruns signifies how many process cycles missed the time constraint set by the period.

5.8.4. Quad Cortex Comparison

The last two signal graphs are designed to compare the Jetson's performance to the Quad Cortex. These fully utilise not only the four available lanes of the Quad Cortex, but also the entire processing power the Jetson can bring to bear in a 128 frame period.
The Quad Cortex is able to process signal graph in $\sim 4.9ms$ which, after subtracting the $\sim 1.6ms$ hardware latency, results in a pure processing time of $\sim 3.3ms$. At 128 frames, the Jetson faster at a mean execution time of $2.3ms$ and a maximum execution time of $2.7ms$ for 99% of the measurement cycles. However, the Jetson is running at full capacity while the Quad Cortex is only at 42% CPU load.

The second graph features a two channel guitar pipeline with dual amplifier and IR processing. Similar to the previous graph, this one fully utilises the Quad Cortex's four lanes and the Jetson's resources. At 128 frames, the Jetson processes the graph at a mean execution time of $2.5ms$ and a maximum execution time of $2.7ms$ for 99% of the measurement cycles. For this graph, the Quad Cortex is two 16-frame periods faster at $\sim 4.2ms$ round-trip latency and $\sim 2.7ms$ pure processing time, achieving a perfect draw with the Jetson. Unfortunately, the Quad Cortex again barely uses half of its processing power at 41% CPU load.

The Quad Cortex is clearly superior with regards to the number of captures it can handle in total. However, I assume this stems more from a more efficient inference implementation than from having more processing power. I am convinced that a custom feedforward implementation for the Neural Amp Modeler ConvNet would allow the Jetson to process more instances.
The Jetson is clearly superior with regards to the flexibility of the signal graph. The Quad Cortex is limited to a maximum of four lanes. While it is definitely possible to max out its CPU load meter, this limit and the restrictions on splitting and mixing between lanes have already annoyed me on multiple occasions. Not having to constrain myself to either parallel or serially focused signal graphs is a huge advantage.
Lastly, the Jetson is actually faster at processing the first graph than the Quad Cortex. Going into this thesis, I did not expect that this proof of concept would be able to outperform a dedicated audio processing unit.

6. Conclusion

The result of this thesis is a proof of concept for an embedded audio processing platform running on the GPU of an NVIDIA Jetson Orin NX. It is fast and powerful enough to handle real-time audio processing at period sizes as low as 32 frames. Unfortunately, even when using the CUDA graph API, there are occasional xruns due to unexpected processing delays within CUDA. At times, the statistics of the performed experiments reported maximum execution times of over 20ms. As a result, the system can not guarantee a WCET low enough to be classified as real-time capable.

Setting this aside, the signal graph size and complexity that this ~900€ chip can handle is impressive. At 32 frames, the Jetson can process the workload of a 24-channel mixing console and, at 128 frames, that of a 64-channel mixing console. When compared against the Quad Cortex, currently priced at ~1600€, the Jetson was faster at processing a four-channel guitar signal chain. The results clearly show that the GPU is well suited to become the core of the customizable and scalable type of audio processing platform that I envisioned.

The developed software already offers more signal routing flexibility than the Quad Cortex. In contrast, the number and quality of the audio effects are not yet comparable. However, with the straightforward `IGpuFx` interface, implementing new audio effects is a simple and fast process.

6.1. Future Work

There are many ways in which this proof of concept can be improved. However, the following three are the most important:

Before anything else, a solution for extreme outliers has to be found. Without guaranteeing a real-time WCET, the GPU will never be used in live sound engineering. To this end, the software should be tested on a dedicated real-time OS to completely rule out the possibility of scheduling interference from other processes.
Next, a dedicated hardware platform and custom audio driver should be developed to reduce the latency introduced by components outside of the signal graph as much as possible. The current setup has a **Jack Loopback Latency** of $\sim 3.5ms$ at 32 frames and $\sim 11.5ms$ at 128 frames. Requiring four latency periods for just the hardware and driver is inefficient, especially at a 128-frame period. The Quad Cortex, in comparison, has a latency of $\sim 1.5ms$ for a comparable signal path at a period size of 16 frames. While running the Jetson at a 16-frames period does not provide a performance benefit, this still shows that there is room for improvement on the hardware and driver side.
Finally, a UI should be developed to allow the user to create and modify signal graphs at runtime.

These three steps are required to turn this proof of concept into a working prototype.

6.2. Outlook

Although I have yet to achieve the goal of creating a real-time capable GPU audio processing platform, I remain convinced that the future of audio processing lies with a general-purpose parallel processing device. If the GPU turns out to be unsuitable, another device will eventually fill the gap.
Presenting the possibilities a real-time capable GPU could bring to the audio processing market is another significant step towards making this shift inevitable.
I will keep working on this project and look forward to contributing towards a new era of audio-processing platforms.

7. Acknowledgements

I am grateful to my supervisor, Prof. Dr. Matthias Rosenthal, for allowing me to pursue my passion for audio processing in this thesis and for his guidance throughout the process.

Furthermore, I am thankful to the ZHAW InEs HPMM team for their advice and support.
Special thanks go to my colleagues: Gianluca Pargätzi for setting up the Jetson Orin NX, Amin Mazloumian for helping with neural network inference, and Marcel Wegmann for his insights into CUDA and for proofreading this thesis.

8. Declarations

This thesis has been written using Github Copilot for improved LaTeX auto-completion and Grammarly for spell-checking and grammar suggestions.

Bibliography

[1] M. Rosenthal et al. *Audio Processing in OpenCL*. Tech. rep. Institute of Embedded Systems ZHAW, 2017.

[2] S. Schneider. *Survey of State of the Art Audio Processing Platforms with AI Support*. Tech. rep. Institute of Embedded Systems ZHAW, 2024.

[3] Goran Nikolic et al. "Fifty years of microprocessor evolution: from single CPU to multicore and manycore systems". In: *Facta universitatis - series: Electronics and Energetics* 35 (Jan. 2022), pp. 155–186. DOI: 10.2298/FUEE2202155N.

[4] NVIDIA Corporation. *NVIDIA AMPERE GA102 GPU ARCHITECTURE*. Tech. rep. NVIDIA Corporation, 2020.

[5] Cong Do et al. "A novel warp scheduling scheme considering long-latency operations for high-performance GPUs". In: *The Journal of Supercomputing* 76 (Apr. 2020). DOI: 10.1007/s11227-019-03091-2.

[6] Nvidia. *NVIDIA Jetson Orin NX Series Modules*. Accessed: September 2024. 2023.

[7] Silicon Labs. *USB Device Firmware Upgrade (DFU) Bootloader Implementation*. Accessed: September 2024. Unknown. URL: https://www.silabs.com/documents/public/application-notes/AN295.pdf.

[8] XMOS. *Fundamentals of USB-Audio*. Accessed: September 2024. 2015. URL: https://www.thewelltemperedcomputer.com/Lib/Fundamentals-of-USB-Audio_1.0.pdf.

[9] Manouane Caza-Szoka and Daniel Massicotte. "Low Group Delay Interpolation Filter For Delta-Sigma Converters". In: *2020 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*. 2020, pp. 1–6. DOI: 10.1109/I2MTC43012.2020.9128668.

[10] Nvidia. *Cuda Programming Guide*. Accessed: September 2024. 2021. URL: https://docs.nvidia.com/cuda/archive/11.4.1/pdf/CUDA_C_Programming_Guide.pdf.

[11] Alexander Brandt et al. *KLARAPTOR: A Tool for Dynamically Finding Optimal Kernel Launch Parameters Targeting CUDA Programs*. 2019. arXiv: 1911.02373 [cs.DC]. URL: https://arxiv.org/abs/1911.02373.

[12] Nvidia. *Cuda Runtime API*. Accessed: September 2024. 2021. URL: https://docs.nvidia.com/cuda/archive/11.4.1/pdf/CUDA_Runtime_API.pdf.

[13] Kishore Kothapalli et al. "A performance prediction model for the CUDA GPGPU platform". In: Dec. 2009, pp. 463–472. DOI: 10.1109/HIPC.2009.5433179.

[14] Shuai Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 2009, pp. 44–54. DOI: 10.1109/IISWC.2009.5306797.

[15] R. Gareous and L. Garrido. *JACK Latency Tests*. https://wiki.linuxaudio.org/wiki/jack_latency_tests. Accessed: 2024-09-30. 2014.

[16] lester michael and boley jon. "the effects of latency on live sound monitoring". In: *journal of the audio engineering society* 7198 (Oct. 2007).

[17] M. Rosenthal. *NVIDIA Jetson Orin NX Modular Vision System*. https://blog.zhaw.ch/high-performance/2024/03/11/nvidia-jetson-orin-nx-modular-vision-system/. Blog post. 2024.

[18] Imgur User. *Image of the front and back of a Focusrite Scarlett 8i6 3rd Gen*. https://imgur.com/nKwfphF. Image. 2020.

[19] Engineering Productivity Tools. *Productivity Tools for Engineers: T001/PT10*. https://web.archive.org/web/20180312110051/http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM. Accessed: 2024-09-23. 2018.

[20] Aaron van den Oord et al. *WaveNet: A Generative Model for Raw Audio*. 2016. arXiv: 1609.03499 [cs.SD]. URL: https://arxiv.org/abs/1609.03499.

[21] Fisher Yu and Vladlen Koltun. "Multi-Scale Context Aggregation by Dilated Convolutions". In: May 2016.

[22] Julius O. Smith III. *Introduction to Digital Filters with Audio Applications*. Accessed: 2024-09-12. 2024. URL: https://www.dsprelated.com/freebooks/filters/.

[23] Balazs Bank. "Converting Infinite Impulse Response Filters to Parallel Form Tips & Tricks". In: *IEEE Signal Processing Magazine* 35.3 (2018), pp. 124–130. DOI: 10.1109/MSP.2018.2805358.

[24] Wikipedia. *Digital biquad filter*. https://en.wikipedia.org/wiki/Digital_biquad_filter. Accessed: 2024-09-30. 2015.

[25] Robert Bristow-Johnson. *Audio EQ Cookbook*. https://www.w3.org/TR/audio-eq-cookbook/. Accessed: 2024-10-02. 2014.

List of Figures

List of Tables

Glossary

ADC Analogue-to-Digital Converter. A device that quantifies the voltage of an audio signal into digital values. 6, 7

ALSA Advanced Linux Sound Architecture. A software framework and part of the Linux kernel that provides an API for sound card device drivers. 14

API Application Programming Interface. A set of subroutine definitions, communication protocols, and tools for building software. 2, 10–12, 14, 15, 17, 18, 20–23, 26, 28–31, 33, 36, 37, 39, 40, 52, 56, 58, 66, 72

C2C Complex-to-Complex. A type of Fast Fourier Transform that transforms complex numbers to complex numbers. 20, 44

C2R Complex-to-Real. A type of Fast Fourier Transform that transforms complex numbers to real numbers. 44

CLI Command Line Interface. A text-based interface used to interact with a computer program. 13, 14

ConvNet Convolutional Neural Network. A class of deep neural networks, most commonly applied to analyzing visual imagery. 23, 24, 63, 65

CPU Central Processing Unit. The part of a computer that performs the instructions of a computer program. Within the scope of this paper, the term CPU is used to refer to general-purpose CPUs like an Intel Core i7 or an AMD Ryzen 9. 2–4, 6, 8, 9, 21, 22, 30, 35, 59, 65, 74

CUDA Compute Unified Device Architecture. A parallel computing platform and application programming interface model created by Nvidia. 2, 4–6, 9–14, 17, 20–23, 27–31, 35, 37, 39, 40, 42, 48, 50, 52, 53, 55, 56, 58, 60, 66, 71, 72

DAC Digital-to-Analogue Converter. A device that converts digital values into an analogue audio signal. 6, 7

DAW Digital Audio Workstation. An electronic device or application software used for recording, editing, and producing audio files. 2

DSP Digital Signal Processor / Processing. Refers to processing digitalized signals or a specialized microprocessor chip, with its architecture optimised for the operational needs of digital signal processing. 2, 3, 7, 8, 15, 71

EQ Equalization. The process of adjusting the balance between frequency components within an electronic signal. 8, 47, 58

FFT Fast Fourier Transform. An algorithm that computes the discrete Fourier transform of a sequence. 20, 21, 28, 44, 55, 58, 63, 64

FIR Finite Impulse Response. A type of filter that has an impulse response that becomes exactly zero after a finite number of samples. 25, 47, 55

frame A measure of the size of multi-channel buffers in audio processing with variable channel counts. Each frame contains one sample of all channels at the same sample / column index. 6, 8, 14, 16, 17, 20–22, 26–28, 30–33, 47, 49, 50, 52, 53, 58, 60, 61, 63–66, 71, 72

GPC Graphics Processing Cluster. A group of texture processing clusters. 4, 5

GPU Graphics Processing Unit. A specialized processing unit optimised to process embarrassingly parallel algorithms like image processing. Within the scope of this paper, the term GPU is used to refer to general-purpose GPUs like a Jetson Nano or a Nvidia RTX 390. 2–4, 6, 9–11, 13, 17, 18, 20–23, 25, 27, 28, 30, 35, 36, 48, 59, 63, 66, 75

GUI Graphical User Interface. A type of user interface that allows users to interact with electronic devices through graphical icons and visual indicators. 9, 14, 22

I2C Inter-Integrated Circuit. A multi-master, multi-slave, single-ended, serial computer bus. 7

I2S Inter-IC Sound. A serial bus interface standard used for connecting digital audio devices together. 7

IFFT Inverse Fast Fourier Transform. An algorithm that computes the inverse discrete Fourier transform of a sequence. 20, 21, 56

IIR Infinite Impulse Response. A type of filter that has an impulse response that does not become exactly zero after a finite number of samples. 25, 35, 47

IR Impulse Response. The response of a filter, system or physical space after having been excited using a dirac impulse. 2, 8, 11, 13, 21, 23, 25, 28, 34, 43, 44, 56, 57, 63–65, 72

NAM Neural Amp Modeler. An opensource VST plugin capable of modeling the distortion characteristics of an amplifier in real-time. 23, 24

ONNX Open Neural Network Exchange. An open-source format for AI models. 23, 24, 36, 46, 71

OOP Object-Oriented Programming. A programming paradigm based on the concept of "objects", which can contain data in the form of fields, and code in the form of procedures. 35, 49

OS Operating System. The software that supports a computer's basic functions, such as scheduling tasks, executing applications, and controlling peripherals. 9, 66

PCB Printed Circuit Board. A board that mechanically supports and electrically connects electronic components using conductive tracks, pads, and other features. 7

PCIe Peripheral Component Interconnect Express. A high-speed serial computer expansion bus standard. 6

period The number of frames that are processes in a single process cycle in real-time audio processing. Can also refer to the capture/playback duration a buffer of this size requires at a given sample rate. 6, 8, 14–17, 19–22, 24, 28, 30, 33, 37, 39, 42, 46, 47, 49, 50, 55, 58–61, 63–66, 71, 72

PFD Partial Fraction Decomposition. A method of decomposing a rational function into a sum of simpler fractions. 25

R2C Real-to-Complex. A type of Fast Fourier Transform that transforms real numbers to complex numbers. 44

RIR Room Impulse Response. The reverberant characteristics of an enclosure captured by recording the response to a dirac impulse. 44

RMSD Root Mean Square Deviation. A measure of the average difference between two data sequences. 26, 49

SDK Software Development Kit. A set of software tools and programs provided by hardware and software vendors that developers can use to create applications for specific platforms. 9, 13

SIMD Single Instruction, Multiple Data. A type of parallel computing architecture that performs the same operation on multiple data points simultaneously. 2, 5, 9

SM Streaming Multiprocessor. A group of streaming processors. 4–6, 10, 29, 55

SP Streaming Processor. Groups multiple functional units with a register file, instruction cache and scheduler/dispatch unit. 4–6, 10, 11

TPC Texture Processing Cluster. A group of streaming multiprocessors. 4–6

UI User Interface. The point of human-computer interaction and communication in a device or software. 8, 12, 34, 36, 37, 39–41, 66

USB Universal Serial Bus. A standard that defines cables, connectors, and communication protocols for connection, communication, and power supply between computers and electronic devices. 6, 7, 14, 15, 18–21, 37, 50

VST Virtual Studio Technology. A software interface standard by Steinberg Media Technologies that allows audio plugins to be used in digital audio workstations. 16, 23

WCET Worst-Case Execution Time. The maximum length of time the system can take to execute a particular task. 37, 66

xrun Buffer underrun. A condition where the buffer is not filled in time, causing an audio glitch. 6, 11, 21, 30, 39, 60, 61, 63, 64, 66, 72

A. Code Snippets

A.1. FxConvFd2c1 Process Method

```
/// @brief convolves an stereo src signal with a mono ir signal
/// @param stream the stream to launch or record work into
/// @param dst the destination buffer
/// @param src the source buffer
/// @param capture_status specifies capture status of stream
/// @return the stream
cudaStream_t _process(cudaStream_t stream, float* dst, const float* src, cudaStreamCaptureStatus capture_status) {
    cufftSetStream(_plan_c2c, stream);

    // pack stereo channels into complex struct and perform fft on both
    IMemCpyNode::launchOrRecord1D(_sig_conv, src, sizeof(float2), _n_proc_frames, cudaMemcpyDeviceToDevice, stream, _src_node, capture_status);
    // forward FFT
    cufftExecC2C(_plan_c2c, _sig_conv, _sig_conv, CUFFT_FORWARD);

    // multiply stereo src and stereo copied IR
    f2f2f2_multiplyAndScale<<<4, 768, 0, stream>>>(_sig_conv, _sig_conv, _ir_fft, _fft_size, 1.0f / _fft_size);

    // inverse FFT
    cufftExecC2C(_plan_c2c, _sig_conv, _sig_conv, CUFFT_INVERSE);

    if (_force_wet_mix) {
        // if force_wet_mix is true, the residual mix kernel can write directly to the output
        IKernelNode::launchOrRecord(1, _n_proc_frames, 0, (void*)f2f2f2_pointwiseAdd, new void*[4]{&dst, &_sig_conv, &_residual, &_n_proc_frames}, stream, _dest_node, capture_status);
    } else {
        // add residual to convolution and write to dst buffer
        f2f2f2_pointwiseAdd<<<1, _n_proc_frames, 0, stream>>>(_wet, _sig_conv, _residual, _n_proc_frames);
        // mix dry and wet signal and write to dst buffer
        IKernelNode::launchOrRecord(1, _n_proc_frames, 0, (void*)f2f2f2_mix, new void*[5]{&dst, &src, &_wet, &_n_proc_frames, &_mix_ratio}, stream, _dest_node, capture_status);
    }

    return stream;
}
```

A.2. Hybrid Graph Building vs. Custom Node Wrapper

```
// Mono to stereo buffer copy using a custom multi-memcpy node
IMemCpyNode* mem_cpy_node;
IMemCpyNode::launchOrRecordMulti(MultiMemcpyType::Segmented2Interleaved, _ir_stereo, &_ir_mono, sizeof(float), _fft_size, 2, {0, 0}, cudaMemcpyDeviceToDevice, stream, mem_cpy_node, capture_status);

// Mono to stereo buffer copy using the hybrid graph building approach
cudaGraph_t graph;
cudaGraphNode_t* nodes = new cudaGraphNode_t[2];
cudaStreamCaptureStatus capture_status_;
const cudaGraphNode_t* dependencies;
size_t n_dependencies;
gpuErrChk(cudaStreamGetCaptureInfo_v2(stream, &capture_status_, nullptr, &graph, &dependencies, &n_dependencies));
cudaMemcpy3DParms params_left{
    .srcArray = NULL,
    .srcPos = make_cudaPos(0, 0, 0),
    .srcPtr = make_cudaPitchedPtr(_ir_mono, sizeof(float), sizeof(float), _fft_size),
    .dstArray = NULL,
    .dstPos = make_cudaPos(0, 0, 0),
    .dstPtr = make_cudaPitchedPtr(((float*)_ir_stereo) + 1, 2 * sizeof(float), sizeof(float), _fft_size),
    .extent = make_cudaExtent(sizeof(float), _fft_size, 1),
    .kind = cudaMemcpyDeviceToDevice};
gpuErrChk(cudaGraphAddMemcpyNode(nodes, graph, dependencies, n_dependencies, &params_left));
gpuErrChk(cudaGraphMemcpyNodeGetParams(nodes[0], &params_left));
cudaMemcpy3DParms params_right{
    .srcArray = NULL,
    .srcPos = make_cudaPos(0, 0, 0),
    .srcPtr = make_cudaPitchedPtr(_ir_mono, sizeof(float), sizeof(float), _fft_size),
    .dstArray = NULL,
    .dstPos = make_cudaPos(0, 0, 0),
    .dstPtr = make_cudaPitchedPtr(((float*)_ir_stereo) + 1, 2 * sizeof(float), sizeof(float), _fft_size),
    .extent = make_cudaExtent(sizeof(float), _fft_size, 1),
    .kind = cudaMemcpyDeviceToDevice};
gpuErrChk(cudaGraphAddMemcpyNode(nodes + 1, graph, dependencies, n_dependencies, &params_right));
gpuErrChk(cudaGraphMemcpyNodeGetParams(nodes[1], &params_right));
gpuErrChk(cudaStreamUpdateCaptureDependencies(stream, nodes, 2, 1));

cufftSetStream(_plan_c2c, stream);
cufftExecC2C(_plan_c2c, (cufftComplex*)_ir_stereo, _ir_fft, CUFFT_FORWARD);
```