GENERAL-PURPOSE GPU AUDIO BENCHMARK FRAMEWORK

Travis Skare *

CCRMA
Stanford University
travissk@ccrma.stanford.edu

ABSTRACT

Acceleration of audio workloads on generally-programmable GPU (GPGPU) hardware offers potentially high speedup factors, but also presents challenges in terms of development and deployment. We can increasingly depend on such hardware being available in users' systems, yet few real-time audio products use this resource.

We propose a suite of benchmarks to qualify a GPU as suitable for batch or real-time audio processing. This includes both microbenchmarks and higher-level audio domain benchmarks. We choose metrics based on application, paying particularly close attention to latency tail distribution. We propose an extension to the benchmark framework to more accurately simulate the real-world request pattern and performance requirements when running in a digital audio workstation. We run these benchmarks on two common consumer-level platforms: a PC desktop with a recent mid-range discrete GPU and a Macintosh desktop with unified CPU-GPU memory architecture.

1. INTRODUCTION

For over fifteen years, commodity graphics cards have contained generally-programmable unified shader pipelines. GPUs excel at highly parallel tasks and operate akin to single-instruction, multiple-data (SIMD) processors. While modern consumer CPUs may have four to sixteen cores, GPUs may have hundreds or even thousands–though GPU cores are much simpler, have more limited instruction sets, operate in clustered groups, and are less suited to applications with dense if-else branching. For certain highly parallel tasks, however, code may be written in an approachable C-like language, and once tuned can reach throughput of ten or one hundred times greater than a CPU implementation.

Researchers have demonstrated feasibility of faster-than-real-time audio processing on the GPU. Early work included the million sinusoids demo of Savioja et al.[1]. This example processed successive time-series steps in parallel; later generations of GPUs were fast enough to process audio sample-by-sample at audio rates. Despite this, there have been few commercial products leveraging the GPU, and most of those had their GPU acceleration phased out. Two examples are earlier versions of LiquidSonics's *Reverberate Core* (convolution reverb) and Acustica Audio's *Nebula 3* (Volterra kernel-based processor). More recently there is the active work of GPU Audio Inc. We suggest possible reasons for the limited market impact include, but are not limited to:

Heterogeneous User Setups: CPU-based plugin developers contend with different processors, vectorization APIs, etc., but may write to a single framework. On the other hand, there are at least three major GPGPU programming frameworks, with different optimal frameworks per GPU manufacturer. While there are some established tools that cross-compile such as Adobe Halide[2] or PyTorch[3], these are more for batch and graph processing respectively and are not as performant as hand-tuned code.

CPU and Input/Output (I/O) Speed Advances: Since the initial release of NVIDIA's CUDA[1] framework, consumer CPUs have moved from around two cores to eight or more, and increased single-core performance substantially. Memory and I/O bandwidth has greatly increased. Modern systems have sufficient CPU power and headroom for most audio production tasks, though GPU acceleration may enable novel applications.

Greenfield Space: CPU-based plugins are a known quantity. GPU-based plugins have less documentation, require additional testing costs, and bring less history in terms of forward- and backward-compatibility. On the other hand, first movers in the space may benefit from the GPU being underutilized.

Cross-product communication concerns: Expanding on the prior point, there are years of history of DAW manufacturers tuning scheduling and thread assignment code for CPU-based effects. The GPU driver coordinating multiple kernel launches is a new system to interact with. A new class of problems is introduced when running multiple GPU-based audio effects from collaborating or competing vendors; this challenge was touched on in prior work[4].

Requires operating in two niche domains: Developers must learn background and best practices in both audio effects processing and GPGPU programming. This increases the "activation energy" needed to get started. However this is approachable with many great resources available. The challenge mirrors that of developing for FPGAs, external hardware, etc.

Toward assuaging some of these concerns, we develop a benchmark framework that can be used to qualify a platform as suitable for GPU-accelerated audio. Such benchmarks may demonstrate that low-level tasks are tractable on a system with time to spare for computation, and may be combined to perform relevant synthesis and processing tasks at audio rates. Then, they may be used to compare different systems and determine the effect of varying parameters such as sampling rate.

Section 2 formally proposes this benchmark framework, establishing metrics, tests, and functionality that simulates the cadence of requests and requirements imposed when running inside a digital audio workstation. Section 3 presents results from running these tests on two modern consumer-level systems typical of what hobbyist or professional music producers may use. We note

* Thanks to conference organizers, hosts, and reviewers.

[1] Compute Unified Device Architecture, the API and tools enabling running non-graphics programs on NVIDIA GPUs

trends on each platform and point out where each excels. Section 4 provides instructions for how to obtain and run the framework.

2. PROPOSAL

A proposed set of benchmarks may be used to answer questions such as:

- Can this system process audio on the GPU at real-time rates?
- What is the overhead? What are the bounds on buffer size and sample rate?
- How many tracks can be processed in real-time? What are the data transfer limits?
- Does this platform excel at or struggle with certain sub-tasks? Perhaps one platform has arithmetic throughput but slow memory access once working set exceeds some limit.

Other work has sought to measure GPGPU performance for certain systems, often datacenter or deep learning systems. Most relevant to audio production is the work of Renney et al.[5] who discuss the feasibility of GPGPU acceleration of real-time audio tasks in particular. Their work provides microbenchmarks and a synthesis benchmark for CUDA and OpenCL, an AMD-compatible API. We note our benchmark suite has overlap with Renney et al.'s microbenchmark section in terms of measuring kernel execution times and data transferring tasks, so we may comment on commonalities and differences between the two studies. In this work we introduce multiple new audio domain benchmarks, study an additional consumer platform with different memory performance, and work toward simulating the conditions of running in a digital audio workstation environment.

Gregg et al.[6] discuss benchmarking general GPU-accelerated tasks. They suggest performance analyses must measure the total end-to-end performance; reporting GPU code throughput in isolation can misleading as it is only part of the overall task. Pre- or post-processing portions of the task may take a substantial fraction of the total wall-clock time and outweigh a hundredfold gain obtained by processing on the GPU. This scenario is an application of Amdahl's law[2]. Gregg et al. also suggest bucketing benchmarks in terms of input-heavy, output-heavy, and data-light benchmarks, and suggest noting whether data transfer may be hidden via asynchronous transfers, or whether it is in the critical path. While we do not use their exact system, and we focus only on audio processing, we take inspiration to benchmark different mixes of input/output sizes, noting audio plugin tasks may be grouped as in Table 1.

Table 1: *Input/Output Skew Cases*

I/O relative sizes	Example effect types
Input « Output	Synthesizers
Input $\simeq$ Output	Console emulation, Guitar amp sim
Input » Output	Metering, Analog summing emulation

[2]Amdahl's law [7] states the maximum obtainable speedup for a task will be limited by the fraction of total time the optimized section takes. If a task consists of 90% parallelizable section and a 10% serial section, we will never be able to exceed a ten times speedup even if reducing the former to zero.

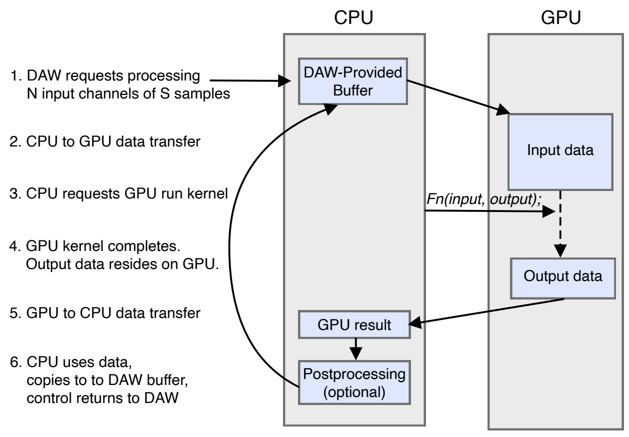
CPU
GPU
1. DAW requests processing N input channels of S samples
2. CPU to GPU data transfer
3. CPU requests GPU run kernel
4. GPU kernel completes. Output data resides on GPU.
5. GPU to CPU data transfer
6. CPU uses data, copies to to DAW buffer, control returns to DAW
DAW-Provided Buffer
Input data
Fn(input, output);
Output data
GPU result
Postprocessing (optional)

Figure 1: *Steps in a GPU-accelerated plugin process*

2.1. GPU-Accelerated Audio Processing

A GPU-accelerated plugin often comprises the following steps, shown visually in Figure 1.

1. The host DAW sends the plugin a block of N input channels of data to process. Example: requests to process N*512 samples arrive every 11 milliseconds.
2. CPU code copies from this block of data into a region of GPU memory; the CPU is called the *host* and the GPU is called the *device*. This host-to-device transfer is either performed with an explicit API call or implicitly using some syntactic sugar or a helper class.
3. With the data to be processed residing on the GPU, the CPU requests the GPU run a *kernel* of one or more functions.
4. The GPU executes the kernel code either synchronously or asynchronously. It writes outputs to memory on the device.
5. Control returns to the CPU which performs an explicit or implicit device-to-host memory transfer.
6. The CPU uses the generated or transformed data. In the case of a plugin, the data would be copied back to the output buffers provided by the DAW.

All benchmarks included in the suite at the time of this writing assume the code under test would be called in such an environment.

2.2. Metrics

We measure two core metrics: latency and throughput.

Latency is the wall-clock time in milliseconds between when an audio buffer is requested to be processed and when its results are available. This is our most important metric: if we miss the deadline for processing a single buffer, a recording take or a song may be ruined. We will report median, 95th percentile, and maximum audio buffer processing times, and care about tail latencies a great deal. In cases of excessive latency metrics, developers may add buffering and leverage the DAW's latency compensation. Batch processing tasks are less sensitive to latency, of course. We may say an audio processing task "meets the latency bar" if it is able to complete within the time allowed for its audio callback. This is defined as the audio buffer size divided by the sampling rate, minus some time for overhead.

Throughput measures how much data a system be processed over time, for example one may compute 500 Megabytes of data per second for some task A and 1,200 Megabytes per second for some task B. This is useful in considering how many tracks of 32-bit audio at a target sampling rate we can process, and when properly measured yields the overall speedup factor over a CPU process.

These two metrics are directly dependent when scaling parameters such as number of tracks and buffer size. Other metrics are not measured directly but may be obtained via profiling tools. In particular this suite of benchmarks has a small working set and small impact on RAM, and memory usage totals are not collected.

2.3. Parameters

The benchmark harness provides tunable parameters for: sampling rate, audio buffer size, and number of virtual tracks. Default values for throughput-based benchmarks simulate processing 64 tracks of ten seconds of 48kHz audio, 512 samples at a time. GPUs excel at higher levels of parallelism, so users are encouraged to experiment with higher track counts. Some benchmarks have additional controls. These may extend the aforementioned parameters (e.g. input/output weight shift for the I/O microbenchmarks) or replace them (number of modes for the modal synthesizer replaces track count).

2.4. Structure

The benchmarks run as a console application. This approach mirrors some prior projects which successfully used a dedicated utility process to handle GPU calls. Future work might add the ability to run benchmark code in-process inside an actual plugin. No third-party translation layer is used; the benchmarks are rewritten for each platform. We use shared superclasses when possible to minimize repetition and remove some API calls from the main path of reading the code.

The inner kernel code is C-like for both Metal and CUDA; debugging and performance tuning per-platform will likely be more time-consuming than migrating syntax. Benchmarks are divided into three sections: initialization, kernel execution, and validation. Some of these sections may be uninteresting in some benchmarks. Notably, all memory allocations should be performed during initialization. For dynamic sets of objects (e.g. synthesizer voices), we preallocate a buffer that can store our maximum number of voices. This suggestion mirrors best practice for real-time CPU plugin development. Validation might compare output to a golden sample, or check that output statistics meet certain criteria. Validation is recommended to ensure all work was performed, and that results match a known CPU implementation.

Next, we discuss the suite's individual benchmarks.

2.5. Benchmarks: Microbenchmarks

Microbenchmarks measure raw capabilities of a system, each concentrating on an aspect such as I/O or arithmetic throughput. These should measure individual sections of a GPU-accelerated audio processor. These include:

Kernel launch: simply launches an empty "no-op" kernel. This measures GPU API overhead and provides a lower bound for a system's feasible audio buffer size (assuming no buffering in the plugin).

Memory transfer: Transfers data to and from the GPU. This is repeated for small and large amounts of data. This is also repeated for input-output balances (as in Table 1) to see if transfers are faster in one direction.

Gain / Metering: These tasks perform a small amount of processing on a large block of samples: adjusting gain and gathering some statistics about our audio (mean, max per channel). We map one virtual audio track to one thread, as this is the most immediate approach to porting existing code to the GPU. This benchmark helps analyze single-GPU-thread performance between systems, rather than total throughput.

2.6. Benchmarks: Macrobenchmarks

This category comprises medium-sized benchmarks which implement domain-specific tasks such as equalization and synthesis which may make up part of a full GPU-accelerated product.

IIR Filtering: Enough filtering for a 5-band equalizer per virtual channel; involves a practical number of multiplies and adds. This is currently implemented as several biquad filters per channel. Stateful systems require persisting that state across kernel launches. There are two approaches to this problem: we may reserve some space in GPU memory to save and restore the state, or we may return it back to the CPU alongside the output data and re-send it to the GPU on the following kernel launch alongside the input data. As audio plugin developers will take the opportunity to optimize their end products, we leave such decisions up to the author of these benchmarks when implementing for a new platform. For the provided CUDA implementation, we round-trip the small amount of state to and from the CPU rather than storing it in GPU memory, as the amount of data is much smaller than the generated audio. This decision would require experimentation for large amounts of look-back and feedback, however. The Metal implementation uses unified memory, so the state can remain in place accessible to the CPU and GPU.

Modal Filter Bank: Modal Synthesis[8] determines resonant modes of a system and sums the output of oscillators or resonant filters tuned to those modes to model strings, bars, and other vibrating objects.

We synthesize N modes; as N may be high, we suggest the results are tree-summed down by a factor of 32, so the benchmark measures processing time rather than data transfer. The state may be saved on the GPU or sent back and forth at the developer's preference.

Here we send audio for the first 64 modes and all state from the device to the host as output, and re-send state back to the GPU on next invocation. This puts more of an emphasis on computational throughput over I/O.

RndMemN: Reads from a block of memory with N virtual playheads; an example use case is a granular synthesizer. A goal is to stress or "bust" caches which are often limited on the GPU, incur penalties for unaligned and unsynchronized memory accesses between threads in an execution group, and incur penalties for using global discrete memory, which based on experience in prior projects was not directly addressable at audio rates (compared to registers, thread-level memory, and shared memory).

Convolution: The Convolution benchmark promotes use of a platform's specialized read-only memory, which might provide higher throughput over non-constant memory. This optimization

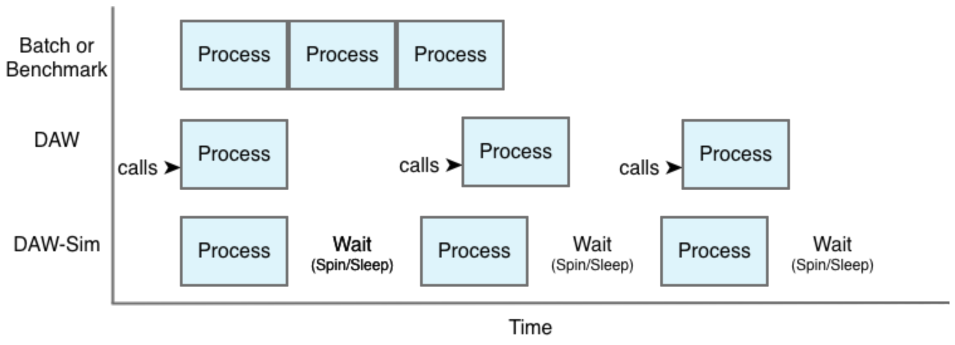

Figure 2: *Batch vs Real-World vs DAW Simulation*

is inspired by Chapter 7 of *Programming Massively Parallel Processors: A Hands-on Approach, 4e*[9]. CUDA/Metal declare read-only buffers as "constant" and "texture" memory respectively.

2.7. Future benchmarks

Please see the repository (links in Section 4) for additional benchmarks over time; the Digital Waveguide Mesh[10] in particular is under development to cover stencil-like memory access patterns. This synthesis pattern extends the digital waveguide to a multi-dimensional grid of bidirectional delay units and scattering junctions, and is an efficient method of synthesizing membranes.

2.8. Closer to real-world: DAW Simulation

Meeting latency bars on the micro- and macro- benchmarks is a necessary but not sufficient step in qualifying a system for GPU audio acceleration. In the real world, a plugin's processing function is called by a DAW or driver many times a second and must service each request before some deadline. Giving the benchmarks access to all data at once is similar to batch processing, specifically benchmarking a DAW's offline "bounce" command. This is a valid user flow, but not the common real-time processing case.

Choosing p95/maximum latency as a core metric addresses this somewhat, but we still miss phenomena such as the OS scheduler treating a bursty high-CPU batch task differently than a long-running periodic processing task.

Batch-benchmarking also experiences contention with other resources differently. The GPU may (or may not) compete with GUI rendering for the OS, DAW, or web browser on the system. Our longer-lived, lower-GPU-usage tasks might have caches on the GPU cleared more frequently, the usage pattern may influence the GPU driver/scheduler as well.

To mitigate this, we introduce a DAW Simulation, or "DAW-sim" for short, benchmark mode that invokes the code under test at regular intervals. The basic concept is demonstrated in Figure 2[3]. To implement this functionality, we must have our processing thread wait for the next audio buffer and have a choice of sleeping vs. spinning on the benchmark thread. Sleeping involves requesting the operating system pause execution of a thread until it is needed or for a certain time period. Spinning involves actively waiting inside a loop: `while(!wait_time_elapsed){}`. This keeps the audio thread awake but costs system resources. Sleeping was measured to have a more significant impact on benchmark runtime; we do not have a deep knowledge of DAW internals but using Windows tracing tools while two mainstream DAWs[4] processed audio suggests sleep system calls were not used for the audio thread; general practice suggests avoiding sleeping on the audio thread as well. We note that in a real DAW environment, other plugins in the same chain will actively spend cycles, potentially making spinning more accurate for dense audio projects. Finally, we note that different operating systems may have soft-realtime capabilities and specialized threading calls such as MacOS's Audio Workgroups on version 11 and above; this is an area for future exploration. We choose a spin approach. In practice, the DAW has control over, and a massive influence on, scheduling and threading behavior.

3. RESULTS

We present detailed results for microbenchmarks and three domain-specific benchmarks as mentioned in Sections 2.5 and 2.6 to compare two modern consumer-level platforms for GPGPU audio acceleration. 32-bit floating point samples are used on both platforms.

The two systems under test are described in Table 2. A Windows PC with discrete NVIDIA RTX 4070 is given shorthand label *PC* and an Apple Mac Mini M2 with integrated GPU is given label *AS*. We note MacOS laptops and desktops produced since late 2020 use ARM-based "Apple Silicon" system-on-chip cores (SoCs) with integrated GPUs and memory shared between the CPU and GPU. The differences between a system with a discrete graphics card with dedicated memory versus the M1-M3 chips is illustrated in Figure 3.

3.1. Kernel startup time

The most minimal benchmark measures overhead of launching and running a kernel. A histogram of bare-bones kernel launch times by platform is in Figure 4. For this case only we include a third platform, an older Intel MacOS desktop with an AMD RX 5700XT GPU. This is a discontinued platform, but similar to those which may be in production environments for years to come. From this

[3]Note in this figure that the real-world DAW calls may have varying inter-arrival time due to other plugins; our benchmark implementation results in periodic and regular calls

[4]Cockos REAPER and PreSonus Studio One

Table 2: *Test platforms, Windows PC with RTX 4070 (PC) and MacOS with integrated M2 GPU (AS)*

Platform Shorthand	CPU	RAM	GPU	VRAM
PC	Intel i7-12700	32GB Discrete	NVIDIA RTX 4070	12GB Discrete
AS	Apple M2 Pro (Mac Mini 10 Core)	16GB Unified	M2 16-Core	16GB Unified

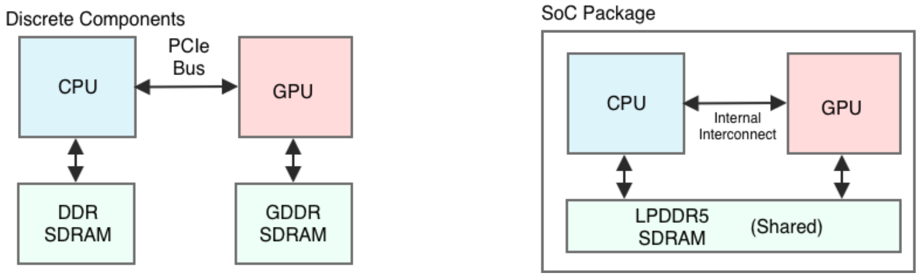

Figure 3: *Common Discrete vs. Unified Memory Architectures*

point we concentrate on a comparison between the two more modern platforms, but the benchmark code does build and run on Intel Macs for developers wishing to support that platform.

Median and 95th percentile latency for each platform is listed in Table 3. We see CUDA has very low kernel launch latencies; Metal's are higher but still fractions of a millisecond.

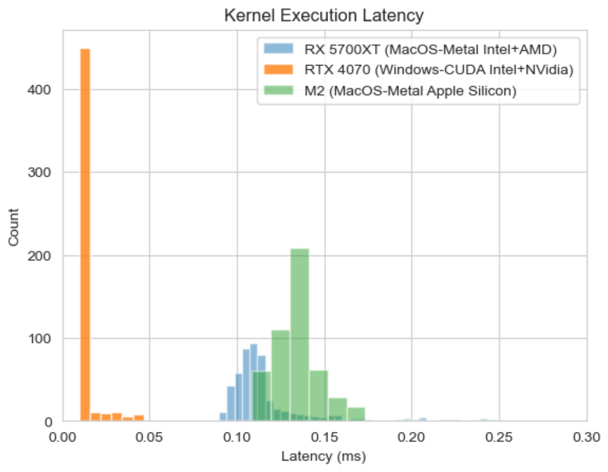

Figure 4: *Histogram of Kernel Launch Times by Platform*

Table 3: *Kernel Launch Time by platform, milliseconds*

Platform	Median	p95
5700XT (Metal/AMD)	0.11	0.17
4070 (CUDA)	0.01	0.03
M2 (Metal/Apple)	0.13	0.17

3.2. Data transfer time

Transferring audio data to and from the GPU for processing is a system-dependent overhead cost. Table 5 shows measurements for scenarios where inputs may be much smaller, similar in size to, or much larger than outputs. We compare outputs for PC and Apple Silicon platforms. For the first four columns, we assume a DAW sends us an input buffer, which we must copy to the GPU, process, and copy back from the GPU to the DAW's provided buffer. The Apple Silicon platform uses shared memory and, if we may process this data in-place, we may avoid these copies. The last two $AS_{unified}$ columns of the table show avoiding these copies reduces tail latencies by 90% or more in this benchmark, suggesting the unified memory architecture may have more time available for arithmetic computation, or can support higher amounts of data.

3.3. Arithmetic: Modal Filter Bank

We synthesize a modal filter bank of up to one million modes using phasor filters[11]. The inner loop is a complex multiplication-based update, stressing the arithmetic units of the GPU. Results are presented in Table 4. We see the discrete GPU has higher arithmetic throughput, as expected for the difference in TDP[5] between the platforms.

Table 4: *Transfer+Kernel execution time (ms) to Synthesize Modes to 512-sample buffer, median and 95th percentiles.*

# Modes	PC_{p50}	PC_{p95}	AS_{p50}	AS_{p95}
1,000	0.050	0.183	0.172	0.203
100,000	0.315	0.447	0.611	0.671
1,000,000	3.168	3.941	5.717	5.87

3.4. Random Memory Access (RndMemN)

In this trial we simulate a sample-based instrument or a virtual granular synthesizer operating on a large sound buffer. As shown in Figure 5, this can present a challenging case for rapid memory access; if we allow threads to have different loop lengths and start positions, virtual playheads' reads from the buffer will not be aligned and performance will be degraded. We default to a 128MiB virtual buffer[6] for this trial.

[5]Thermal Design Power, a measurement that scales with maximum power consumption

[6]1 mebibyte (MiB) = 2^{20} = 1,048,576 bytes; compare with 1 megabyte (MB) = 1,000,000 bytes

Table 5: *100MiB Data transfer times between CPU and GPU: total latency, in milliseconds, 50th and 95th percentiles, for different mixes of input and output sizes. Time measured includes copying to/from audio buffers as a DAW would provide for first four columns. $AS_{unified}$ variants exclude these copies for cases where unified memory can be utilized.*

Input/Output Weight %	PC_{p50}	PC_{p95}	AS_{p50}	AS_{p95}	$AS_{unified,p50}$	$AS_{unified,p95}$
1 / 99	10.06	11.82	9.14	9.25	0.15	0.632
20 / 80	9.84	11.68	8.77	8.85	0.134	0.185
50 / 50	9.57	11.5	8.49	8.56	0.157	0.556
80 / 20	9.37	11.1	7.93	8.01	0.130	0.175
99 / 1	9.09	10.59	7.66	7.75	0.133	0.17

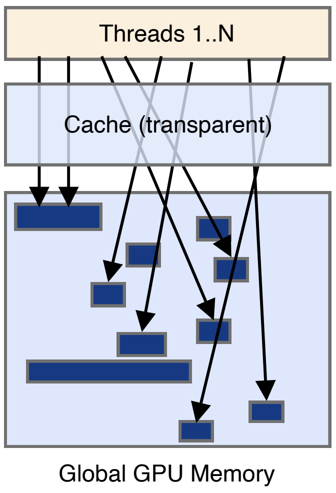

Figure 5: *Virtual wavetable/graintable requiring unaligned and random memory accesses in the degenerate case.*

The buffer is sized sufficiently large, virtual playheads are spaced randomly throughout the buffer, and virtual loops are set to different lengths of up to one second of audio, in order to exceed caches and force reads to the underlying memory. Buffer size is the default 512 samples at 48kHz. We assign one track per GPU thread–a production application might process fewer threads simultaneously to have working set fit in cache. Audio from each individual playhead is returned to the host; a production application might choose to mix some of these on the GPU side to avoid I/O costs isolated in Section 3.2. Results for varying number of virtual tracks/grains are in Table 6. We see the AS architecture performs well in this test; the discrete GPU can still read from the 128MiB buffer at audio rates.

3.5. 1D Convolution

A parallel convolution experiment involves several threads each convolving an input buffer of with its own assigned static impulse response. The impulse response memory may be declared constant or “texture” memory on each platform which may enable performance gains. With 128 threads and impulse responses of length 1024, there was no significant performance gain marking the impulse response buffers constant. On the PC platform, performance analysis revealed the data was cached as it fits in thread-local memory.

Table 6: *Latencies for N tracks reading into 128MiB buffer, milliseconds. Processing 512 samples at 48kHz.*

# Tracks	PC_{p50}	PC_{p95}	AS_{p50}	AS_{p95}
32	0.138	0.197	0.229	0.295
64	0.141	0.218	0.609	0.711
128	0.163	0.427	0.657	0.787
1024	0.364	0.578	0.723	0.784
4096	1.030	2.69	0.719	0.925
8192	1.931	2.253	0.772	1.070
16384	3.810	4.808	0.924	1.218
32768	12.225	20.863	3.618	4.140
65536	19.297	36.334	4.801	7.460

3.6. Impact of DAW-Simulation

For quantitative results of DAW-sim impact, please see Table 8. Results show an often small, but consistent increase in median and tail latency when this mode is enabled. The effect is more pronounced in the compute-heavy benchmark on the Apple Silicon platform. We consider the kernel invocation a bit of an outlier as it is much simpler and quicker than the other benchmarks.

Table 7: *DAW-Sim compensation: Disabled vs Sleep vs Spin, PC platform, 1M Modes Benchmark*

Metric	Disabled	On-Sleep	On-Spin
p50 latency (ms)	2.87	7.37	2.87
p95 latency (ms)	3.73	10.94	8.07
Max latency (ms)	3.96	11.07	10.96

Table 8: *Impact of enabling DAW-sim flag on p50 latency; runtime increase over baseline*

Test	$PC_{off \to on}$	$AS_{off \to on}$
Kernel Invocation	2.740	2.863
100MiB I/O 1/99	1.049	1.108
1GiB I/O 1/99	1.021	1.033
100MiB I/O 99/1	1.002	1.090
1GiB I/O 99/1	1.028	1.005
100 Modes	1.000	1.537
1M Modes	1.093	1.283

Next steps in this area would be to run the benchmark code as plugins inside a DAW and log latency stats. We may also wish to run other plugins or processes alongside our console app to increase contention, ideally in a repeatable manner.

3.7. Observations

These benchmark results allow us to make observations about the two platforms under consideration. The kernel launch and data transfer microbenchmarks show that both platforms are able to transfer significant amounts of data to and from the GPU during an audio callback, with the majority of available callback time remaining for processing. These results mirror the microbenchmarks in Renney et al.[5], who also measured function execution latency and data transfer time, and obtained similarly successful results on an NVIDIA and an AMD platform.

The higher-level, domain-specific benchmarks demonstrate the systems' ability to execute practical real-time audio effect tasks. Both platforms were able to synthesize several thousand modes or sound grains at audio rates. The desktop GeForce GPU excels at raw arithmetic throughput, with correspondingly higher power draw, while the unified memory architecture on the Apple Silicon system allows us to eliminate significant data transfer overhead and use a larger proportion of our audio callback for computation.

4. OBTAINING AND RUNNING THE CODE

The framework may be found mirrored at the following URLs:

- https://github.com/tskare/gpuaudiobench
- https://cm-gitlab.stanford.edu/travissk/gpuaudiobench

The repository contains independent and separately-buildable PC (C++ / CUDA) and MacOS (Objective-C / Metal) implementations. The PC implementation was developed and run on Windows, but OS-specific functionality was avoided.

Usage instructions and available benchmarks may be found in the corresponding `README.md` files or by specifying `-help`. Global system parameters such as sampling rate or DAW simulation mode may be adjusted in the code, with commandline parameters forthcoming.

Each benchmark will run several times (default 100), and collect median, 95th percentile, and maximum seen latency. The console will print these latency statistics, and write them to a small statistics file for easier consumption by scripts. A human-readable message will also be printed, noting if that trial consistently met real-time latency requirements. For example, a successful run might output:

```
    OK: max latency 1.212 ms under 10.667 ms
callback time limit.  Please consider a
margin of safety as well.
```

In case the benchmark code had latency values that would have overrun an audio callback, the message will be of the form such as:

```
    WARNING: p95 latency 46.367 ms over
10.667 ms callback time limit (median ok).
```

5. CONCLUSIONS

A benchmark framework for GPU-Accelerated Audio was presented. This includes microbenchmarks and domain-specific benchmarks. A novel extension to the benchmark framework towards simulating the environment of data processing inside a digital audio workstation was implemented and quantified.

Benchmark code was run on two contemporary, mid-range, consumer MacOS and Windows systems. Both platforms were revealed to be suitable for real-time audio processing. We note high arithmetic throughput on the PC with discrete GPU and the benefits of a unified memory architecture for reducing data transfer times.

6. REFERENCES

[1] Lauri Savioja, Vesa Välimäki, and Julius O Smith III, "Real-time additive synthesis with one million sinusoids using a GPU," in *Audio Engineering Society Convention 128*. Audio Engineering Society, 2010.

[2] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand, "Halide: Decoupling algorithms from schedules for high-performance image processing," *Communications of the ACM*, vol. 61, no. 1, pp. 106–115, 2017.

[3] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala, "PyTorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019.

[4] Travis Skare, "GPGPU patterns for serial and parallel audio effects," in *Proceedings of the 23rd International Conference on Digital Audio Effects (DAFx-20)*, 2020, pp. 125–131.

[5] Harri Renney, Tom Mitchell, and Benedict R Gaster, "There and back again: The practicality of GPU accelerated digital audio.," in *NIME*, 2020, pp. 202–207.

[6] Chris Gregg and Kim Hazelwood, "Where is the data? why you cannot debate CPU vs. GPU performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE, 2011, pp. 134–144.

[7] Gene M Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[8] SN et al. Hou, "Review of modal synthesis techniques and a new approach," *Shock and vibration bulletin*, vol. 40, no. 4, pp. 25–39, 1969.

[9] W Hwu Wen-Mei, David B Kirk, and Izzat El Hajj, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, 2022.

[10] Scott A Van Duyne and Julius O Smith, "Physical modeling with the 2-d digital waveguide mesh," in *Proceedings of the international computer music conference*. International Computer Music Association, 1993, pp. 40–47.

[11] Max Mathews and Julius O Smith, "Methods for synthesizing very high Q parametrically well behaved two pole filters," in *Proceedings of the Stockholm Musical Acoustics Conference (SMAC 2003)(Stockholm), Royal Swedish Academy of Music (August 2003)*. Citeseer, 2003.