GPGPU PATTERNS FOR SERIAL AND PARALLEL AUDIO EFFECTS

Travis Skare

CCRMA
Stanford University
Stanford, CA, USA
travissk@ccrma.stanford.edu

ABSTRACT

Modern commodity GPUs offer high numerical throughput per unit of cost, but often sit idle during audio workstation tasks. Various researches in the field have shown that GPUs excel at tasks such as Finite-Difference Time-Domain simulation and wavefield synthesis. Concrete implementations of several such projects are available for use.

Benchmarks and use cases generally concentrate on running one project on a GPU. Running multiple such projects simultaneously is less common, and reduces throughput. In this work we list some concerns when running multiple heterogeneous tasks on the GPU. We apply optimization strategies detailed in developer documentation and commercial CUDA literature, and show results through the lens of real-time audio tasks. We benchmark the cases of (i) a homogeneous effect chain made of previously separate effects, and (ii) a synthesizer with distinct, parallelizable sound generators.

1. INTRODUCTION

General-Purpose GPU programming is attractive to obtain substantial speedups over CPU implementations for applicable problems. GPUs have been used in 2D and 3D FDTD methods to bring complex simulations from >1000x real-time to 10-50x real-time. For example, Bilbao et al.[1] simulate a timpani drum membrane and the volume of air around the instrument. A C implementation obtains 4x speedup over a MATLAB prototype while GPGPU CUDA code obtains 30x speedups over the prototype.

Real-time applications exist. Belloch et al.[2] demonstrate wave-field synthesis with multiple objects via fractional delays, and Webb[3] leverages multiple GPUs to compute room acoustics.

Commercial applications are rare but do exist. The previous version of Acustica Audio's Nebula[4] had a CUDA bridge to accelerate their proprietary Volterra kernel-based algorithms, though support for the GPU engine does not currently exist in the newest version. Recently, NVidia released "RTX Voice" which offers real-time noise cancellation, targeted for voice streaming scenarios.

Some challenges exist in widespread adoption of GPU audio. From a casual survey of online forums, wide differences in users' setups and worries about I/O bandwidth and latency (especially latency *variance*) are common concerns. Practically, CPU performance has continued to increase year over year and consumer core counts grow. The number of audio signal paths is closer to CPU core count (low dozens) than GPU core count (low thousands).

Copyright: © 2020 Travis Skare. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

While certainly not the only hurdle, we will focus on another consideration–coordinating multiple GPU "programs." The GPU can be shared between multiple processes and the operating system (for compute or display), but context switches introduce system overhead. Furthermore, several GPU programs that each copy data to and from the card must share I/O capability, and may quickly run out of time in the audio callback. GPU manufacturers offer documentation and tools for optimization, so in the following sections we apply a few such strategies to synthetic benchmarks and observe performance increases, with the constraint that we must meet audio rates.

As an aside, there is some academic coverage studying the wisdom of using GPUs for real-time tasks outside graphics applications. Yang et al.[5] provide interesting observations and benchmarks for embedded GPUs architecturally similar to our system. In addition to tips and experiments, they point out concerns with incomplete or incorrect documentation. While the group's field is autonomous vehicles, where the failure modes can be more severe than an audio dropout, the concerns may be of interest to audio researchers as fellow real-time system developers.

1.1. GPU Terminology

We provide a brief overview of some GPU facts and terms relevant for future sections.

CUDA[6] is NVIDIA's General-Purpose GPU (GPGPU) development platform for GeForce cards from the last decade. It provides APIs for critical tasks such as transferring data between CPU and GPU memory, and executing a GPU "kernel" (a function-formatted block of program code).

The developer may specify how many threads run a given kernel. Regardless of this choice, threads are grouped and scheduled in blocks of 32 called a *warp*. If the developer requests 31 threads, one warp will be used and one thread will be turned off. If the developer requests 33 threads, two warps will be used, one full and one to compute the 33rd thread. Warps contain some shared resources that make sharing data inside this group quick. On the other hand, threads in a warp may compete for resources such as the arithmetic units or memory bandwidth.

The GPU handles scheduling of warps based on available resources. The GPU may schedule a kernel execution work in multiple stages transparently, and may serve requests from other processes and the operating system on its own schedule.

Memory access patterns can be critical in obtaining performant code. While beyond the scope of this work (specifics will be different for different algorithms), the development kit includes debugging functionality to assist in optimization.

1.2. Setup

Next, we describe optimizations for cases of combining effects in serial and parallel, and test these via synthetic benchmarks. The test system is an Intel 3570K CPU with 16GB of RAM and an NVIDIA GeForce 1080Ti, which is a powerful consumer-level card, though currently three generations old (released in March 2017). The GPU has 11GB of RAM, runs at a 1480 MHz processing clock, and has floating-point throughput of 10609 GFLOPS at single precision, 332 at double precision. In practice many applications will not be constrained by these limits and will be more sensitive to latency. Neither the system CPU nor memory capacities will bottleneck us in these studies. Memory and I/O speeds will play a part and more modern and high-end hardware are expected to have more performant numbers[1], but similar scaling trends.

We mix development on Linux and Windows; for this work Windows was used the majority of the time due to preference for the development tools available on that platform. Of note, GeForce/CUDA drivers are not available on MacOS 10.14+. In the event that this changes, we expect the benchmarks to run there as well. CUDA Toolkit 10.1 was used for this work.

Benchmarks take the form of measuring time to compute ten seconds of audio under the parameters. Sampling rate is held at 44.1kHz, and number of effect channels is held at 64 unless specified otherwise–this is less parallelism than many GPGPU algorithms can reach, but it is a realistic thread count when treating an audio channel as our unit of parallelism.

We run benchmarks for two classes of virtual insert plugin: "Light Plugin" and "Heavy Plugin." Both plugins accept a buffer of audio and need to process it. Light runs the equivalent of ten biquad filter calculations per sample of audio; Heavy runs one thousand, requiring 100x the FPU throughput. Thus, the Light plugin models cases where I/O time dominates CPU time, and the Heavy plugin case models cases where CPU time dominates I/O time.

2. HIDING DATA TRANSFERS

First, we consider a documented strategy for increasing throughput when running one single homogeneous audio task, by reducing I/O time. This will be also generalized to multiple heterogeneous kernels.

A prior project[7] involved a GPU process to synthesize and modulate many modes in parallel. The GPU synthesis/filterbank process and a plugin client running on the same machine inside a DAW are launched and begin communicating over shared memory[2]. Then, during each audio callback, the following steps happen:

1. The plugin parses MIDI input data and populates a region of shared memory with filter coefficients for the filter bank process.
2. The plugin signals semaphore A and waits on semaphore B.[3]
3. The filterbank process has been waiting on semaphore A; it wakes up and transfers filterbank coefficients to GPU
4. The filterbank process launches the GPU synthesis kernel and waits.
5. The GPU kernel retrieves prior state from main GPU memory (device memory), synthesizes a buffer of new samples, and saves new state back to main GPU memory.
6. The GPU kernel threads synchronize and sum their output buffers in groups of 32 threads.
7. The output buffers are transferred back to the filterbank process (host memory).
8. If host memory is not mapped cross-process, the samples are copied to shared memory that *is* mapped cross-process.
9. The GPU signals semaphore B and waits on A for the next buffer.
10. The plugin wakes up and copies output data to the location provided by the DAW. It returns and will be invoked to process the subsequent audio callback.

In the existing implementation, all tasks happen sequentially and strictly serially; they do not overlap. However, GeForce cards from the last several years support concurrent memory copy and code execution. That is, we can pipeline our code: while we are copying results from kernel execution N, we may be in the compute phase for execution $N + 1$, and midway through this we may have input data for execution $N + 2$ available and we can start copying that to the device. This strategy was used by Belloch et al.[8] towards 16-channel massive convolution on the GPU.

A hypothetical case we will use to benchmark this optimization is shown in Figure 1. The baseline approach reads the next available input audio segment, copies it to the graphics card, invokes the kernel, and copies output data from the card to the host process. The pipelined architecture uses asynchronous versions of the API to overlap execution execution and I/O. We modified the code to double-buffer inputs and outputs, though this change is not absolutely required if the host is guaranteed to process output data while I/O is not in progress.

One caveat: for zero-latency plugins, we must compute the three stages (transfer-to-device, kernel execution, transfer-from-device) serially and within one audio callback. We note Figure 1 would fail to meet this constraint even with the performance optimization. There is a more general issue here: because input samples for callback N+1 must be sent to us after we deliver data for callback N, the opportunity for overlapping data transfers disappears and therefore this optimization can't be applied for zero latency plugins. If we are willing to incur one or two buffers of latency, however, this optimization may expand the set of algorithms that are practical on a GPU. We can effectively hide latency, run compute near 100% of the time, and transfer results back to the DAW in time slice after the one where compute finishes. In contrast, the serial, non-pipelined example in Figure 1 will never be able to keep up with audio rate.

This is benchmarked synthetically in Table 1. We transfer buffers of N samples of 64 channels of audio to the GPU, run a computation representing a single monolithic insert plugin with five internal stages of either "light" or "heavy" processing detailed above. This processes samples serially using the arithmetic units of the GPU. We then transfer the buffers back to the host.

When plugins are computationally light but still parallel, we see improvements of around 15% by hiding I/O; this is a good

[1] For reference, the more recent flagship, GeForce 2080 Super, has a 1650 MHz base clock and 10138 single-, 317 double-precision GFLOPS

[2] without loss of generality; these processes could even live on different machines if latency allows.

[3] We note for production-quality software, we must allow other dependencies to run while we wait, so we must write threading/locking code responsibly.

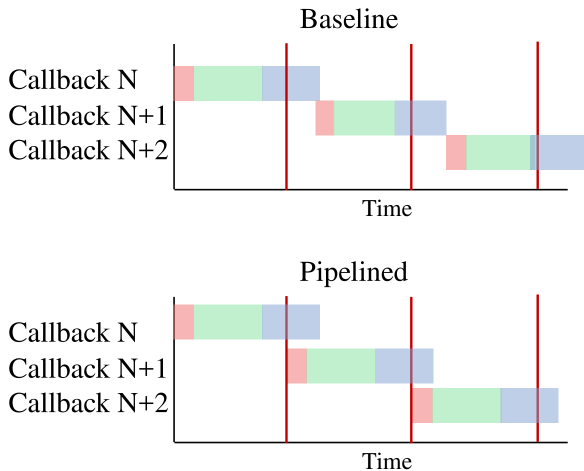

Figure 1: *Pipelined approach using concurrent execution and memory transfer. Vertical lines indicate audio callback deadlines. Processing segments are colored red for host-to-device I/O, green for GPU kernel execution, and blue for device-to-host I/O.*

Table 1: *Sequential vs. Pipelined execution for one GPU-accelerated plugin. Time to process 10 seconds of audio (sec)*

Strategy	Light Plugin	Heavy Plugin
Baseline (N=256)	2.86	16.14
Pipelined (N=256)	2.43	16.32
Baseline (N=1024)	1.17	4.88
Pipelined (N=1024)	0.98	4.43

outcome-to-effort ratio for simply moving to the asynchronous API. We still see improvement in the case of more computationally-intensive kernels, but the improvement is small in marginal terms, as there is relatively less I/O to hide.

The preceding benchmark measures processing throughput. Digital Audio Workstations also have real-time constraints. In Table 2 we rerun the experiment and measure mean, variance, and maximum buffer processing times as seen by the host process. As in the throughput-focused ten-second processing tests, these measurements include all I/O transfers from and to the GPU. Overhead of transferring to and from a host DAW is *not* included.

Table 2: *Sequential vs. Pipelined execution, buffer processing time, in milliseconds.*

Strategy	Mean	Variance	Max
Baseline (N=256), Light	1.19	1.82	9.04
Pipelined (N=256), Light	1.00	1.56	10.09
Baseline (N=1024), Light	2.11	1.79	11.08
Pipelined (N=1024), Light	1.93	1.63	8.46
Baseline (N=256), Heavy	8.49	0.91	12.77
Pipelined (N=256), Heavy	8.36	0.89	12.35
Baseline (N=1024), Heavy	8.76	1.10	16.38
Pipelined (N=1024), Heavy	8.61	1.31	15.31

Practically, at 256 samples at 44.1kHz, we must process a buffer of audio in less than 5.6ms, minus applicable margins and overheads. We note this constraint is not met for maximum latencies in Table 2 for this trial. This is a problem for the baseline case, though in pipelined cases our deadline will be two or more audio callbacks, so the deadline may be met (of course this presents as higher user-audible latency).

3. VIRTUAL CONSOLE SERIAL EFFECTS CHAIN

We consider a hypothetical 64-channel virtual console emulation. This might be modeled using a series of stages:

- Microphone preamp: trim/gain and nonlinearity simulation.
- Channel Equalizer
- Channel Compressor

A simple approach is to execute kernels serially for each stage, copying output data to a buffer to be used as the input data for the next module. This is analogous to the way insert plugins may be chained in digital audio workstations.

We narrow the problem with the restriction that each audio channel is processed with one thread, sample-by-sample. In making this restriction, we note that many GPU algorithms have additional parallelism we may wish to exploit where possible. In fact, the earliest examples of GPU audio computation on modern graphics pipelines depended on this to reach audio rates. Savioja et al.[9] sums more than one million sinusoids on a GPU, but required computing multiple samples for the same sinusoid in parallel. Yet some algorithms cannot exploit time parallelism; in these cases, $f(t+1)$ has a dependence on $f(t)$ and we cannot apply any strategy to change that. Based on experimentation, modern GPUs have clock rates, I/O bandwidth, floating-point throughput high enough to make serial sample-by-sample processing feasible for some applications.

Returning to the virtual console example: we code the benchmark with each stage modeled as a separate effect. Because each kernel invocation by default involves sample transfer between CPU and GPU, we spend a large percentage of wall-time waiting for I/O needlessly. An optimization is to combine our effects chain into a single kernel and compute the full chain without any unneeded memory copies. This scheme is visualized in Figure 2, and is benchmarked in Table 3 for a chain of k effects.

Table 3: *Coalescing k plugins to avoid I/O transfers between plugins. Time to compute 10 seconds of audio (sec). Buffer size N=256.*

Strategy	Light Plugin	Heavy Plugin
Baseline (k=2)	1.79	3.29
Combined (k=2)	1.05	2.32
Baseline (k=5)	3.42	7.51
Combined (k=5)	1.09	3.74
Baseline (k=10)	6.79	14.75
Combined (k=10)	1.16	6.22

The benchmarks show eliminating I/O is, as expected, a very useful optimization. For inexpensive but parallel GPU computation kernels, I/O dominates scaling. As we chain computationally-inexpensive plugins we quickly use up our audio buffer timeslice in data transfer. If we can optimize all but the first and last transfers away, we can scale number of plugins, or plugin complexity, much further and spend more time in compute code versus I/O.

We again run the benchmark to examine practical latency considerations, in Table 4 the mean latency, variance, and maximum

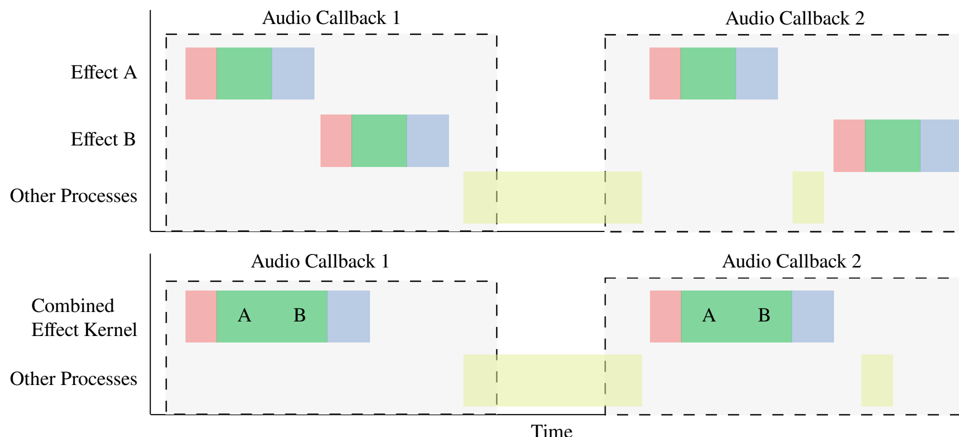

Figure 2: *Combining serial effects into one kernel*

latency are measured. The more time spent inside compute-heavy regions, the less penalty we suffer from inefficient I/O, though the optimization is still worth pursuing.

Table 4: *Coalescing k plugins to avoid I/O transfers between plugins. Buffer processing time, milliseconds. Buffer size N=256*

Strategy	Mean	Variance	Max
Baseline (k=2), Light	2.09	2.39	10.94
Combined (k=2), Light	1.24	1.55	9.19
Baseline (k=5), Light	4.96	3.90	23.58
Combined (k=5), Light	1.52	1.47	8.66
Baseline (k=10), Light	10.18	5.32	33.22
Combined (k=10), Light	1.85	1.35	9.58
Baseline (k=2), Heavy	16.76	1.22	21.18
Combined (k=2), Heavy	15.74	0.82	19.48
Baseline (k=5), Heavy	41.56	1.46	45.99
Combined (k=5), Heavy	38.09	0.71	41.27
Baseline (k=10), Heavy	83.25	1.92	89.58
Combined (k=10), Heavy	74.31	1.25	83.37

Combining effects plugins is a simple operation, but we have a few options that balance code readability, flexibility, and resource usage.

As a manual approach, we may simply merge code in our development environment, or more elegantly merge via preprocessing, the code from two kernel functions into a new, combined function. Here we must ensure that variables are not redeclared, and that the correct buffers are used such that the output memory from one plugin is used as the input to the next. A compile-time transform could be written for this. One drawback is that each plugin will reserve its own stack space and potentially registers for variables. This is the approach used for our benchmarks in this work.

Alternatively, we may break our code into multiple `device` functions, with a new `device` function calling them serially, but developer documentation suggests that in cases where the compiler does not inline the functions, the stacks will use relatively slower device memory, where we would ideally like to keep all locals in registers if possible; in [7] this was one of the most attractive optimizations to running certain algorithms at audio rates. We note we may reuse locals between stages at the cost of readability; beyond this we hold discussion on local variables for Section 4.1.

In a hypothetical case where we are unable to coalesce all effects into one kernel, as a middle ground we may keep separate kernels, but add a boolean parameter that controls whether to copy data back to the host, or merely keep it in global GPU memory. This parameter must be true for the last effect in the chain, but the prior effects would be instructed to leave data in global memory and avoid sending intermediate buffers back.

It is worth noting some advantages of separate kernels: clean separation of code, separate stacks, easy code reuse across projects, the ability to easily chain multiple instances (for example a fast-attack compressor to tame peaks followed by a "character" compressor for tone), and the ability to easily modify order – compressor before EQ rather than after. While one monolithic kernel does not preclude any of those advantages, when writing kernels separately the advantages are obtained "for free."

4. SYNTHESIZER WITH PARALLEL MODULES

Consider a semi-modular polyphonic synthesizer where oscillators may be chosen from different engines: for example, FM or digital waveguide string simulation. The simplest way of structuring this code is likely to write it the same way we might a CPU implementation, performing a `switch`-like selection of algorithm:

Listing 1: *Default approach for a multi-generator synth*

```
device synthKernel1(args) {
  // Local variables declared here
  // for both engines

  if (config[threadIdx.x].synthEngine
        == FM) {
    // run FM for this thread
  }
  else if (config[threadIdx.x].synthEngine
        == STRING) {
    // run string simulation for this thread
  }
}
```

Note that `threadIdx` is provided by the CUDA runtime; each thread will see its own index in this expression.

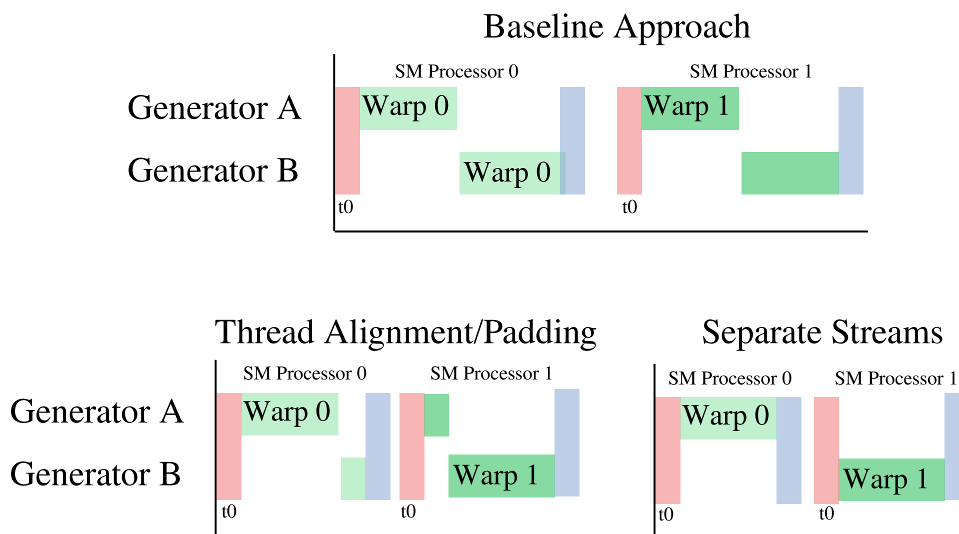

Figure 3: *Default behavior and two optimizations for a synthesizer with selectable generator module. Note that the two warps are scheduled to execute simultaneously on different streaming multiprocessors.*

We recall how contemporary GPUs handle branching: for a given `if()` branch in code, the conditional is evaluated on all threads in a warp. If any thread in the warp evaluates to true, the branch is taken. All threads in the warp execute in lockstep; the threads that failed the condition effectively wait and do not incur side effects. In an `if/else`, some threads will execute one branch then wait for the other threads to execute the other branch. We note future GPU generations may support richer branching.

Branching can reduce performance by factors of two, especially troubling for deep conditional statements. The GPU, however, will transparently implement an important optimization: if *all* threads in a warp fail a conditional, there is no need to execute it and the GPU will jump past it.

Therefore, if we can structure our threads in a way to maximize the likelihood that all threads in a warp execute the same algorithm, our branches do not incur overhead.

We can group threads by voice and still run the check as in Listing 1, or save some memory and partition the space; threads with IDs less than some constant `ENGINE_A_THREADS` will use generator of type A (FM in the prior example) and the next partition uses generator type B (STRING in the prior example).

Listing 2: *Partitioned voice space optimization*

```
device synthKernel2(args) {
  // Local variables declared here
  // for both engines

  if (thread_index < ENGINE_A_THREADS) {
    // Run code for synth Engine A
  } else {
    // Run Code for synth Engine B
  }
}
```

4.1. Local variables

Performance may be the primary concern, but it is good to keep readability and maintainability in mind. The `synthKernel2` approach in Listing 2 has drawbacks, including more complex code, and more importantly a shared frame for all local variables, imposing a resource constraint on register/thread memory, which is a limited yet valuable resource.

It is possible to have the kernel code call other `device` functions. In this case, the function may either be inlined, or called with a traditional stack architecture; the stack resides in global memory and will not be as fast as registers but can benefit from caching.

CUDA supports a `union` type but seemingly not in registers as of the development kit version used in this project.

4.2. Comparison

A visual summary of the different approaches for the "synthesizer with multiple generator types" problem may be seen in Figure 3.

We have a large number of voices–the synth may be polyphonic and have several oscillators per voice–and we assign generators to CUDA threads in order of their display on a user interface. This is easiest to debug and the simplest, cleanest coding practice. At runtime, though, it is likely that in a group of 32 threads, we'll have some threads using Generator A and others using Generator B.

The second approach, "Thread Alignment/Padding," can use the exact same kernel code, but when assigning threads to warps we assign them by Generator type. "Padding" refers to the fact that we may end up with fewer than 32 threads per warp, and have some threads idle in each warp. If only a few voices uses the FM engine, we may seem to "waste" the majority of a warp, but in practice the tradeoff in throughput may be worthwhile.

We expect some overhead in the Thread Alignment approach: the kernel code is larger. There may be cycles spent in preconditions or calculating the branch predicate for a branch we do not

take. Finally, we may have fewer registers for optimizations[4].

The third approach, "Separate Streams", also involves bucketing threads, but we use the CUDA *Stream* API to launch each generator as a completely separate kernel, concurrently. Benchmarks follow in Table 5.

Table 5: *Parallel Synth Engines, Time to compute 10s of audio (sec)*

Strategy	Medium Plugin	Heavy Plugin
Baseline	2.94	9.03
Thread Alignment	1.87	4.51
Streams	1.95	4.73

We see the expected effect that grouping threads to eliminate branches has a substantial positive effect, approaching 2x for our branching factor under the right conditions. Interestingly, we see simply organizing threads manually performs best with these simple kernels. The Streams API offers true independent kernel execution, but there seems to be some small overhead in coordinating and scheduling multiple kernels. In this synthesizer example it was trivial to group threads; for other cases streams may be worth considering using Streams to overlap heterogeneous kernels.

5. PRACTICAL CONSIDERATIONS

Some practical issues must be addressed to create a usable system for end-user audio engineers. Different GPUs support a different number of concurrently-executing kernels within the same process, which limits the number of different algorithms that may execute simultaneously. The number may be queried from the hardware at runtime or looked up in the developer documentation based on Compute Capability level for the card[6], and is expected to be around 16-32 in theory, with lower practical limits.

The prior section assumed a system with only one process coordinating CUDA kernel launches. For our synthetic benchmark synthesizer we have full control, but what if a competing plugin brand is running their own GPU synthesizer? By default, multiple processes on a CPU system have different CUDA "contexts" and can not overlap. Each process effectively owns a global CUDA context lock for its timeslice. For maximum performance, the community would likely need to maintain a shared GPU effect bridge and kernels could be submitted as bridge plugins in the form of shared libraries, much as DAW plugins are compiled and registered.

An alternative exists, however. Nvidia provides MPS, a "Multi-Process Service" binary which can coordinate kernel executions, and importantly tries to overlap memory and compute tasks between different processes, without those processes being aware of each other.

MPS is beyond the scope of this work. For interested researchers, note Yang et al.[5] benchmark MPS for real-time applications and find some pros and cons; their results suggest this is not a magic bullet. A failed effort to port TensorFlow to support MPS[10] showed that it limits programming power, though MPS is more robust on newer GPU architectures: Volta[5], Turing, and beyond.

[4]This last case would represented by a longer "Warp 0" bar in the plot

[5]Professional cards only; no consumer-level Volta cards were launched

It is worth emphasizing the overall utility of GPU audio is dependent on exploitable parallelism: either we must have an algorithm with parallel pieces per audio stream, or a high number of independent audio streams. Revisiting a FPU-bound "heavy plugin" example from Section 2 that processes ten seconds of audio as quickly as possible, and sending the kernel 64 channels of audio, we find via the NVidia performance tools that we use 0.16% of available streaming multiprocessor throughput and 0.37% of available memory. These are low numbers but make sense: the card has 3,584 streaming multiprocessors and 64 channels of audio only requires four or eight (at common developer-chosen dimension sizes), plus we do not achieve 100% occupancy. Overall memory transfer rate achieved is 1.73 GB/s. The card has a theoretical maximum of 484 GB/s; CPU, Memory, and I/O counters all indicate the 64-channel application is data-starved, as expected.

Having touched on coexisting with other audio tasks and other GPU tasks (OS drawing, other applications, plugins' UI rendering), we attempted to quickly gauge real-world impact. Different benchmarks were run with and without Geeks3D "FurMark" running on the system simultaneously, set to the "GPU Stress Test" mode. A synthetic FPU-bound trial increased in time from 4.0s to 10.71s (2.68x) with FurMark running. For an I/O bound test, the audio processing process's runtime increased from 15.06s to 48.05s (3.19x). Thus, GPU scheduling contention may be an area worth exploring in the future, with regard to different consumer workloads and dedicated versus shared GPUs.

6. CONCLUSIONS

We explored some optimizations available for cooperative GPU kernel execution in serial and parallel real-time audio tasks. Alternative approaches were considered, along with tradeoffs involving development flexibility and style.

Benchmarks were synthetic, but the parallel synthesizer modules strategy is being applied in practical concurrent work involving a drum set synthesizer with a modal filterbank for cymbals and waveguide meshes for membranes.

Very recently, Gaster et al.[11] explored feasibility of different buffer sizes for realtime GPGPU audio, benchmarking across CUDA and OpenCL on discrete and integrated graphics. We concentrated on CUDA and explored breadth in the form of benchmarking different optimizations, and found overall similar possible buffer sizes. However, we note that while buffer sizes of 256 samples appear feasible in terms of mean latency, latency variance, and data throughput, there were occasional latency spikes that would have caused rare, but present, overruns–even before taking into account DAW overheads, driver latency, and additional safety margins.

Future work might be to fully prototype a virtual console setup, if a model is available where GPU compute for 64 channels outweighs I/O transfer. This would give a practical example for the effects chain benchmark.

We might also investigate integrated GPUs on mobile devices. These are becoming steadily more powerful and are often generally-programmable. They are of active interest in the machine learning community; Georgiev et al.[12] found that inference for tasks such as speech recognition obtained 6.5x speedups over CPU while using only 25% of the energy, saving battery power and heat output. Anecdotally, as an early empirical result we have found modal processors may run on the NVidia Jetson Nano development board at audio rates, albeit with thousands rather than millions of modes.

Use of mobile GPUs could power art installations or portable instruments and run at around five watts, or increase the scope of what is possible on phone/tablet synthesis applications.

7. ACKNOWLEDGMENTS

Thanks to reviewers who had substantive comments that improved the paper and allowed for improvements such as formalizing insights around latency, pointers to `nvprof` and GPU stress tests. Thanks to conference organizers, especially during this unique and challenging year.

8. REFERENCES

[1] Stefan Bilbao and Craig J Webb, "Physical modeling of timpani drums in 3d on gpgpus," *Journal of the Audio Engineering Society*, vol. 61, no. 10, pp. 737–748, 2013.

[2] Jose A Belloch, Alberto Gonzalez, Enrique S Quintana-Orti, Miguel Ferrer, and Vesa Välimäki, "Gpu-based dynamic wave field synthesis using fractional delay filters and room compensation," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 25, no. 2, pp. 435–447, 2016.

[3] Craig Webb and Alan Gray, "Large-scale virtual acoustics simulation at audio rates using three dimensional finite difference time domain and multiple graphics processing units," in *Proceedings of Meetings on Acoustics ICA2013*. Acoustical Society of America, 2013, vol. 19, p. 070092.

[4] Acustica Audio, "Nebula 3 vst plugin," 2009.

[5] Ming Yang, Nathan Otterness, Tanya Amert, Joshua Bakita, James H Anderson, and F Donelson Smith, "Avoiding pitfalls when using nvidia gpus for real-time tasks in autonomous systems," in *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[6] NVIDIA Inc., "CUDA C++ Programming Guide," Available at https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, accessed April 19, 2020.

[7] Travis Skare and Jonathan Abel, "Gpu-accelerated modal processors and digital waveguides," in *Linux Audio Conference*, 2019.

[8] Jose A Belloch, Alberto Gonzalez, Francisco-Jose Martínez-Zaldívar, and Antonio M Vidal, "Real-time massive convolution for audio applications on gpu," *The Journal of Supercomputing*, vol. 58, no. 3, pp. 449–457, 2011.

[9] 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.

[10] TensorFlow GitHub repostory and community, "Support for nvidia-cuda-mps-server," Available at https://github.com/tensorflow/tensorflow/issues/9080, accessed April 19, 2020.

[11] Benedict R Gaster, Harri Renney, and Thomas J Mitchell, "There and back again: The practicality of gpu accelerated digital audio," in *NIME*, 2020.

[12] Petko Georgiev, Nicholas D Lane, Cecilia Mascolo, and David Chu, "Accelerating mobile audio sensing algorithms through on-chip gpu offloading," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, 2017, pp. 306–318.