GPU-ACCELERATED MODAL PROCESSORS AND DIGITAL WAVEGUIDES

Travis Skare
CCRMA
Stanford University, USA
travissk@ccrma.stanford.edu

Jonathan Abel
CCRMA
Stanford University, USA
subsection abel@ccrma.stanford.edu

ABSTRACT

Digital waveguides and highly-resonant filters are potential fundamental building blocks of physical models and modal processors. What might a sound designer accomplish with a massive collection of these objects?

When the building blocks are independent, the overall system becomes highly parallel. We investigate the feasibility of using a modern Graphics Processing Unit (GPU) to run collections of waveguides and filters, toward constructing realtime collections of room simulations or instruments made up of many banded waveguides.

These two subproblems offer different challenges and bottlenecks in GPU acceleration: one is compute-bound while the other has memory optimization challenges.

We find that modern GPUs can run these algorithms at audio rates in a straightforward fashion–that is, sample-by-sample without needing to implement transforms that allow computation of subsequent time samples concurrently. While a fully-realized instrument or effect based on these building blocks requires additional processing and will have more data dependencies that reduce parallelism, we find that consumer-GPU-accelerated audio enables a scale of realtime models which would be intractable on contemporary consumer-CPUs.

1. INTRODUCTION

Potential applications for a large number of modal filters or digital waveguides include:

- A large collection of coupled acoustic spaces, for example an opera house with listening booths that may be seen as resonators, or the interior architecture of ancient Chavín[1].
- A virtual orchestra where we have many players, each using an instrument made up of several digital waveguides.
- A virtual reality simulation where a server may track room- and position-dependent modal reverberators for a number of participants on low-power client devices.
- A drum set made up of a couple dozen individual instruments, each using many modal filters.

While the first three ideas are hypotheticals enabled by having access to massively parallel filtering/waveguide systems, the fourth exists as a real-world proof of concept to synthesize a dozen modal cymbal models at realtime rates using a GPU. Active work is toward adding realtime controls for a performer.

1.1. Building Blocks: Modal Synthesis and Digital Waveguides

Modal synthesis involves determining the natural resonant modes of a vibrating object, and using the appropriate frequencies, amplitudes, and decay rates to build a system that simulates the original sound.

A filter bank of high-Q filters is often used for such sound synthesis, and is also the backbone of modal reverberators[2]. The more modes we can compute at realtime audio rates, the higher the fidelity of the sound, and the more sources or rooms we may model.

Digital waveguides[3] are efficient for simulation of traveling waves, and with scattering junctions and nonlinearities added, a wide range of physically-accurate bowed strings, brass, etc. may be simulated with robust realtime performance controls. Here, we are interested in working toward many virtual performers each playing an independent instrument (orchestra), or one performer given control over simultaneous but mostly independent "clusters" of waveguide-powered instruments, such as a virtual drum set with a large number of pieces.

Digital Waveguides may be implemented efficiently in the 1-dimensional case via a bidirectional delay line representing two traveling waves, plus filters to account for dispersion loss. These are the basic building blocks we seek to accelerate, noting that for more complex physical models we will add scattering junctions, additional filtering, and nonlinear elements incurring additional computation cost. In some cases, such as piano string modeling[4], some terms may be commuted, or combined with an impulse response, to add complexity to the overall model without scaling the overall steady-state computational cost.

1.2. GPU Acceleration for Audio Algorithms

For years, graphics processing units (GPUs) have supported both high-level realtime graphics APIs as well as lower-level, general-purpose computational APIs. GPU acceleration of audio synthesis and audio effect algorithms has been shown to yield substantial speedups on certain algorithms. GPUs advance in performance each generation in terms of parallel core count and base core speed, so we expect some previously intractable problems to become tractable over time.

Among papers in the literature:

Savioja et. al.[5] give an overview of potential audio tasks that may be accelerated via GPGPU programming at audio rate and reasonable buffer sizes for realtime performance. Sinusoid-based additive synthesis obtained 250x+ speedup over CPU implementations, FFTs running on a GPU were able to be eight times as long as those running on a CPU-based implementations, and FIR filters were able to be 130 times as long as their CPU counterparts. In [5] and [6], the authors showed it was possible to synthesize 1.9 million sinusoids in realtime, a 1300x speedup over a serial lookup table computation on one CPU. This was on a GPU that is six generations behind ours and three major GeForce architecture revisions behind our card[1]. And we note that our graphics card is itself now a generation and major architecture advance behind the times. This work results in a

[1]Fermi (GTX 480, 2010) $\rightarrow$ Kepler $\rightarrow$ Maxwell $\rightarrow$ Pascal (GTX 1080Ti, 2017); RTX cards released in 2018 use the Turing architecture.

realtime sound canvas with more "paint" than previously available (how might an artist use 1.9 million partials in additive synthesis?). The million sinusoids example also demonstrates that maximum performance requires tuning and knowledge of the specific underlying hardware.

Trebien et. al.[7] use modal synthesis to produce realistic sounds for realtime collisions between objects of different materials. Noting that IIR filters do not traditionally perform well on GPUs, due to dependence on prior state not mapping well to the parallel nature of GPUs, they introduce a transform to change this into a linear convolution operator and to unlock time-axis parallelism.

Belloch et. al. [8] accelerate IIR filters on the GPU directly by using the Parallel IIR representation. They achieve 1256 concurrent 256th-order IIR filters at audio rates and sub-millisecond latency at a 44.1kHz sampling rate.

Subsequently, Belloch covers GPU-accelerated massively parallel filtering in [9], and Belloch et. al.[10] leverage GPU acceleration to implement Wave Field synthesis on a 96-speaker array, with nearly ten thousand fractional-delay room filters with thousands of taps. The maximum number of simulated sound sources is computed for different realtime buffer sizes and space partitions; with a 256-sample buffer (5.8ms at 44.1kHz), between 18 and 198 real-time sources could be placed in the field.

Bilbao and Webb[11] present a GPU-accelerated model of timpani, synthesizing sound at 44.1kHz in a 3D computational space within and outside the drum. The GPU approach uses a matrix-free implementation to obtain a 30x+ speedup over a MATLAB CPU-, sparse-matrix-based prototype, and a greater-than-7.5x speedup over single-threaded C code baseline. The largest (and most computationally-expensive) drum update equation is optimized to 2.04 milliseconds per sample, where the bottleneck is a linear system update for the drum membrane.

Our area of study utilizes recursive filters and unfortunately optimizations of the million-sinusoinds and Parallell IIR filter works do not apply directly; we would like to be able to adjust parameters arbitrarily in realtime and at sample rate, which would require rerunning transformation code too often.

Still, our filter bank is expected to be highly parallel in terms of independence *between* the filters. We may have coupling between modes, but so long as it's limited, we can implement this in a way that is compatible with GPU programming ideas. We also do not need to implement arbitrary IIR filters as in Belloch et. al., but will be able to use special-purpose damped oscillation filters that only require a first-order complex update equation (see Section 2).

If GPUs have advanced enough in terms of increased clock rate, increased floating-point resources, and lower memory latency in the last few generations, we aim to compute filter and physical model updates sample-by-sample in realtime.

1.3. GPU Programming

Next, we present a brief overview of GPU programming, and note advantages and challenges versus programming for a general-purpose processor.

Various toolkits exist to develop GPU programs: two of the biggest are NVIDIA's CUDA for use with their graphics cards, and APIs implementing OpenCL, a more general heterogeneous computational framework. For the following investigation we use CUDA. If readers have any modern NVIDIA card, they may download the software developer kit at developer.nvidia.com.

When starting to port an algorithm to the GPU, we must consider if it has parallelism to leverage. NVIDIA coined the term "single instruction, multiple thread" (SIMT) as a variation on the "single instruction, multiple data" (SIMD) of vector processors and modern mainstream processors. If our work is a series of several different and dependent computations, we may not be able to achieve a speedup. If we can structure it as applying identical operations to many points, it is a good candidate for acceleration.

The core work unit in CUDA is a group of 32 threads, called a *warp*. Each thread in a warp may have its own values for local variables, but all threads in a warp will always run the same instruction simultaneously.

A warp is executed on a *Streaming Multiprocessor* (SM). Different graphics cards have different numbers of SMs; a low-power embedded device may have two while our graphics card used for the trials below has 28.

A trivial example task would be to take N integer inputs and double them.

There are two main steps involved in this task. First, we write a *kernel*, the code that will run on the GPU. This will accept an array of inputs; each thread will index into the array, find the element it is to double, multiply it by 2, and store it in an output array. Second, we write host (CPU) code that calls a CUDA function to send an input array to the GPU, execute the kernel, wait for the kernel to complete, and finally copy the output values back to the CPU, for example so we can save them to disk.

If we have 32 inputs to double, CUDA will execute our kernel code on one warp of 32 threads. All 32 threads in that warp execute in lockstep and run the same instructions, but obtain a different value of the array to double and a different output location to store the result. If we have only 15 inputs to double, this is not a problem. We will still run on one warp, and the 17 threads without any work to do effectively get a break (they technically are issued instructions but do not write to memory or compete for resources). If we have 33 inputs to double, we outgrow one warp. Threads will be grouped in one warp of 32 threads and a second warp of one solitary thread. More than one warp can run at a time, so it is very likely we will run in the same time as it took to run the 32 and 15 input cases.

A CUDA-enabled graphics card has some number of Streaming Multiprocessors (SMs). The product specifications for individual graphics card models and a capabilities table such as provided in the CUDA Programming Guide lets authors know how many threads may be in flight per SM, and how many may actually get run each cycle.

For example, our graphics card may have up to 64 warps assigned to each SM ($\rightarrow$ 64warps $*$ 32threads/warp $=$ 2048threads), though only 4 warps (128 threads) may be scheduled on each single clock cycle.

The programming guide lists other bottlenecks and numbers to consider. One piece of information very relevant to us is the number of simultaneous arithmetic operations available.

The graphics card we use is a consumer card meant for gaming. Some other cards (the NVIDIA TITAN for example) are targeted more for enterprise and scientific computing uses, albeit at a significantly higher price point.

We note the issue rate of floating-point operations from the programming guide:

This means that if we require 64-bit precision, our consumer card is more likely to be bottlenecked by this figure than the enterprise card in the lineup.

On the other hand, we note that our card has higher per-clock

Table 1: *Throughput of FP instructions (Results per Clock per SM)*

	Consumer	Enterprise
16-bit mult+add	2	128
32-bit mult+add	128	64
64-bit mult+add	4	32

32-bit floating-point throughput. Assuming that is sufficient precision for a problem, this means our card may execute 128 32-bit multiply-adds per SM $*$ 28 SMs $=$ 3584 multiply-adds per clock. For reference, the card's core clock runs at 1.3-1.5GHz.

Thus far we've considered parallelism and availability of arithmetic units; we must also keep the memory hierarchy in mind.

Each SM has some number of registers. These are very fast. The compiler will attempt to use them for thread-local variables. On our card, there are 64,000 32-bit registers per SM.

Each *thread block* (user-defined organizational unit of threads, comprised of one or more warps) has some fast "shared" memory. This is 64KB per SM for our setup. We'll return to this later as an optimization.

All threads may access a card-wide pool of read-only constant memory.

All threads may also access a pool of global memory–11GB on our card. However, this is described as having roughly 100x the latency of shared memory or registers.

If a thread's local data will not fit in registers, the compiler may reduce parallelism or spill to "local memory." This technically lives in the slow global device memory pool, but is backed by a cache.

1.4. Development Approach

We take an iterative development approach, getting a basic algorithm working and then proceeding to tune it in stages. The CUDA toolkit contains IDE plugins and debugging tools, making it straightforward to analyze bottlenecks as we encounter them. The compiler will also be helping us along the way.

To set expectations, we know there will be overhead involved in transferring data between CPU and GPU, overheads in starting and stopping our kernel, and overhead introduced by the host operating system. We try to mitigate some of these, but some are unavoidable.

It is also important to note the significant effort that would be involved in moving from this proof of concept to a commercial DAW plugin. A hypothetical DAW is competing for CPU resources, will be using the GPU to render its GUI (our kernels can run alongside that with no issue, but there's still potential resource competition), and will force our choice of buffer size and latency.

1.5. Test Setup

The test setup consists of:

- GPU: An NVIDIA GeForce GTX 1080Ti, which is a consumer-grade graphics card, though a relatively high-level one.
- CPU: An Intel i5 3570K running at stock speed. We note this CPU is six generations old and a mid-level chip even in its generation, and newer CPUs may include newer vector instructions including AVX-512. However it is unlikely to bottleneck us, as it is used primarily for memory transfer and GPU kernel launches.
- RAM: CPU has 16GB, GPU has 11GB; neither will bottleneck us in these synthetic benchmarks.
- Storage: consumer SATA SSDs that will not be a bottleneck, especially since our tests should reside completely in RAM.
- OS and software: Development was cross-platform; kernels were written on Ubuntu Linux with Microsoft's open-source VSCode as a text editor and compiled using the CUDA Toolkit. During the memory optimization phase of the project, NVIDIA Nsight Visual Studio Edition on Windows was used for its "Next-Gen CUDA Debugger," though it is noted that the Linux/Mac Eclipse edition also contains an Eclipse-based profiler.
- Programs were compiled as 64-bit in case we use more than 4GB of RAM, possible with high buffer sizes and high numbers of parallel waveguides.

We discuss development of two algorithms: high-Q filters suitable for use in modal processors, and a simplified form of digital waveguides, running independently without scattering junctions and only a gain multiplier in the feedback loop. These two systems were developed simultaneously and do not depend on each other; we begin with the modal filter code since it is simpler, can essentially ignore the GPU memory hierarchy (everything besides output data fits in registers), and we estimate will be bottlenecked exclusively by the floating-point throughput of the graphics card, which makes it the easier of the two to optimize.

2. MASSIVE MODAL FILTER BANK

As described above, a modal filter bank used for synthesis, effects or reverberation consists of N resonant filters. We make the assumption that all the filters are uniform in construction and vary in parameters; a GPU can of course run multiple styles of filters in parallel, either through conditional execution or simultaneous kernel execution.

In practice, rapidly changing the coefficients on e.g. Direct-Form II filters may result in audible artifacts. In [12] Max Mathews and Julius Smith proposed a filter that is very-high-Q, numerically stable, and artifact-free, based on properties of complex multiplication.

This is suitable for modal synthesis and reverberators such as in [2]; the recursive update equation we need to implement is:

$$y_m(t) = \gamma_m x(t) + e^{(j\omega_m - \alpha_m)} y_m(t-1) \qquad (1)$$

where:
$x()$ is an input or excitation signal.
ω_m is mode frequency m.
γ_m is a per-mode complex input amplitude gain.
α_m is a per-mode dampening factor.

This is straightforward to implement; the state we store for each mode is limited to the prior output $y_m(t-1)$, the parameters α_m, γ_m, and ω_m, even if only for intermediate computation. For simplicity we keep them all; noting that while complex values use two 32-bit registers each (four when using 64-bit precision), we likely have 255 registers per thread and have room to spare.

We benchmark three approaches:

When letting these resonating filters run as undamped oscillators, we are able to compute and reuse the complex exponential value, and only conditionally add the input term; with these simplifications we will require two floating-point multiplies per cycle. We create a benchmark to determine the number of such oscillators we can run in parallel in realtime. We run two variations of this benchmark at different buffer sizes. A third benchmark simulates a pesrformance that modulates all the filters on every sample: we re-compute the exponential term each time it is used, and look at the performance impact.

We move to benchmark those three approaches. In more detail:

Free-Run is the optimal case where the oscillators only need to update based on a complex multiplication of $y(t-1)$ with a static value of the complex exponential. A buffer size of 2,000 samples is likely larger than we'd want for realtime performance (45ms at 44.1kHz), but allows us to reduce kernel-switch overhead.

Small Buffer is identical to Free-Run, but with a buffer of 256 samples (5.8ms at 44.1kHz).

Continuous Modulation is our third approach, simulating gain parameters and frequencies changing continuously, requiring recomputing the complex exponential term with each sample update, in addition to performing the 2-multiply complex update of the filter state. This case uses the same 256-sample buffer as *Small Buffer*.

We measure the amount of time it takes to render ten seconds of 44.1kHz audio for N phasor filters in parallel. This means that benchmark runtimes over 10 seconds fall behind realtime performance, while values under 10 might be feasible. For each trial, the median of three runs was used; in practice we did not see large outliers in these tests.

Tabulated results are in Table 2; bold entries took less than ten seconds to compute and thus are candidates for realtime performance. In practice, we might want to avoid values under but close to ten seconds, due to system variance and unmeasured overhead of a DAW, OSC server, controller processing, etc. The same data is available as a plot in Figure 1, with a horizontal line representing realtime limits. In all graphs in this paper, lines between sample counts are present only to show trends, and we do not expect results for intermediate values of N to fall precisely on that line.

Table 2: *Time to run N filters for 10 seconds of Audio*

N Filters	Free-run	Small Buffer	Continuous Mod.
458752	**1.48**	**2.95**	**3.97**
917504	**2.63**	**4.18**	**5.54**
1835008	**4.85**	**7.17**	**8.49**
3670016	**9.23**	11.39	13.21

Some observations:

As these filters are completely independent, we achieve high utilization on the GPU and are only blocked on availability of floating-point units. All data is stored in registers and we avoid memory accesses, especially global memory accesses.

It is worth reiterating that this is benchmarking building blocks. We synthesize audio and copy it back to the host RAM, but additional logic is needed on the CPU to modulate parameters based on realtime user input or performance data and most likely to post-process the output with effects.

Using a smaller buffer incurs more cost, which can be 50% and even higher, percentage-wise, for low N. At very high N the effect is lower; we bottleneck on floating point unit availability in the large-buffer version, but have lower kernel launch overhead.

As a final observation on Table 2's data, the continuously-modulated version does not suffer as large a performance penalty as expected since it looks like we had some idle 32-bit floating-point units - they are not occupied every cycle. It also allows us to eliminate a conditional check since we always run that logic.

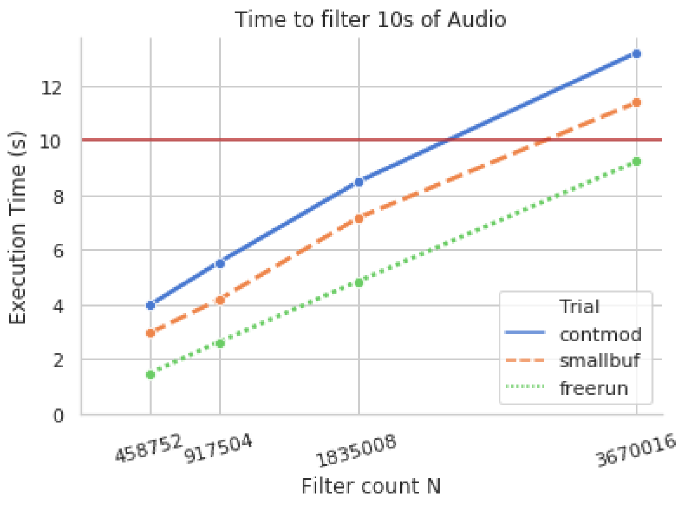

Figure 1: Time to run N filters for ten seconds of samples under different trials.

Moving forward, we benchmark the use of double-precision arithmetic. We made an alternate 64-bit kernel - basically swapping `cuDoubleComplex` in for the default `cuComplex`, which is by default `typedef`ed to be single-precision.

With a 256-sample buffer and continuously-changing parameters, and N=458,752 filters, it takes 19.49 seconds to render 10 seconds of audio. Our corresponding single-precision trial only took 3.97 seconds, so we note a 4.9x slowdown. As noted earlier, each SM on our GPU may only issue four 64-bit floating point multiply-adds versus 128 32-bit adds. As we did not achieve 100% utilization of the floating point units in prior benchmarks, we don't necessarily suffer a 32x (128/4) slowdown, but it is clear we are being bottlenecked by double-precision FPU availability with this configuration.

As we might expect, scaling down to 114,688 filters lowers resource contention enough to run within our realtime constraints (7.04 seconds to synthesize 10 seconds of audio). If we need the extra precision, that is likely still more than enough high-Q filters to enable some interesting instruments and effects, such as creating a virtual drum set with several thousand filters available to each instrument.

3. MASSIVE WAVEGUIDE "ORCHESTRA"

Next, we code up a kernel that performs the computations for a simple 1-D Digital Waveguide. This follows the description of the structure from Section 1.1; each thread owns a bidirectional delay line made up of continuous memory on its thread stack. This is used as a circular buffer, with an index value serving as a read/write head, and a multiplicative factor on feedback introduces dispersion loss. Note that for most waveguide-based physical models, additional code will be needed for scattering junctions, nonlinearities, etc., reducing our maximum throughput and complicating our kernel code, but parallel waveguides may be benchmarked as a starting point, to suggest an upper bound for performance.

As a baseline, we start with uniform waveguides of delay-length M=5000 samples in total[2], and process audio in 2000-sample chunks

[2]lengths in this benchmark represent the total length of the delay in the system; if building a waveguide from a bidirectional delay line, each delay

with each launch of the GPU kernel.

Because a buffer size of 2000 at 44.1kHz would be 45 milliseconds–longer than we'd like for interactive applications–we performed the same trial at 256 samples (5.8ms).

Then, because we expect to run out of thread registers and spill to expensive "local" memory (as above, really in the global pool) we run a second variant of waveguides of length 10 samples and a buffer of 2000 samples–arguably too much of a simplification, but this may be useful to establish a loose upper bound on performance.

In each trial, we compute the amount of time it takes to repeatedly run the kernel on the GPU and copy some data back out to the host.

The data copy is non-negligible overhead; assuming 1.8M waveguides and buffer of length 2000, we generate 13GB of audio data each kernel execution. As such, we first sum all the samples in each warp to reduce the overhead by a factor of 32 - still leaving us with a substantial amount of sound data to transfer across the PCI Express bus.

As in the high-Q filter benchmarks, we build N independent objects in parallel and measure the time it takes to synthesize ten seconds of sound at 44.1kHz.

Results are in Table 3. N is a multiple of 32 to ensure all warps are occupied. Bold entries take less than ten seconds to compute and thus ran faster-than realtime. A plot of the data is in Figure 2.

Table 3: *Time to generate 10s of Audio, Uniform Waveguides*

N DWGs	Baseline	Small Buffer	Short Waveguide
3584	**0.249**	**0.544**	**0.11**
14336	**0.522**	**0.811**	**0.272**
57344	**1.44**	**1.75**	**0.95**
114688	**2.79**	**3.09**	**1.83**
229376	**5.5**	**5.74**	**3.68**
458752	10.85	11.07	**7.28**
917504	21.14	20.24	14.59
1835008	42.611	49.72	29.152

We note some trends:

As expected, computing more waveguides requires more time. Scaling is sub-linear while growing at small N as we utilize more of the GPU in the parallel section of the benchmark ("for free"), but we still incur a cost for memory transfer of the outputs off the card, which itself scales linearly with N. The parallel sound synthesis portion of the program becomes linear with N as resources are exhausted; beyond this point we essentially are cycling through groups of warps serially.

Decreasing the buffer size from processing 45ms to 6ms of audio per kernel execution did not seem to affect the feasible N as much as anticipated. There is a notable 2x difference at small N but for both, the 458,000 waveguides trial was not feasible while the 230,000 waveguide trial used approximately 55% of the available time slice.

A variation of the trial using a shorter waveguide showed that through the range of our trial values of N, scaling is partially dependent on memory usage. As noted above, this is an experiment performed to validate that, as we might expect, longer-length delay lines may incur more computational cost. Of course, the delay line lengths used in practice will be defined by our physical model and sampling rate.

would have length $M/2$

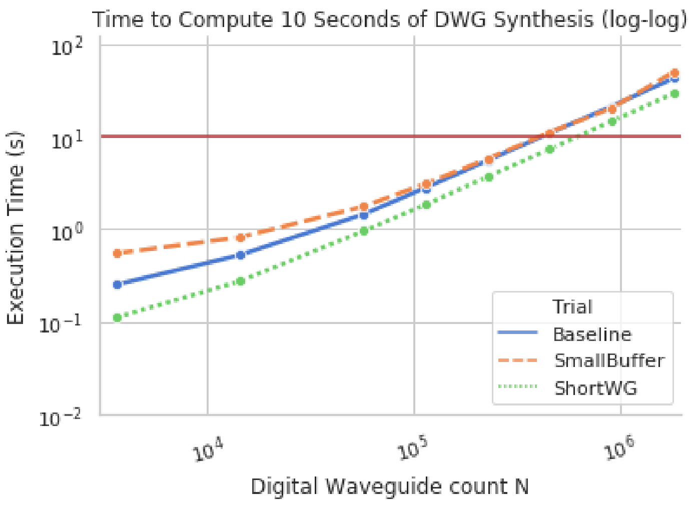

Figure 2: Time to run N waveguides for ten seconds of samples under different trials.

These preliminary benchmarks suggest a rough upper bound, so in our iterative development approach we return to coding, refining our kernel and un-relaxing some assumptions. Currently, all the waveguides have the same delay line length; this is unrealistic for real-world applications, so we next move to have waveguides play one of 1000 different pitches, by having each parallel digital waveguide own a delay line of different length. We use our baseline setup, and allocate the same blocks of memory as before, but have waveguide n be of length $128 + 5n$ mod 5000. This means that all waveguides inside a warp will have different delay line lengths, and warps compute different values overall.

Results are in Table 4.

Table 4: *Time to generate 10s of Audio, "Baseline" uses same-length waveguides, "Differing" experiment uses heterogeneous waveguides. Slowdown Factor is the multiplicative performance penalty.*

N DWGs	Baseline(s)	Differ.Lengths(s)	Slowdown Factor
3584	**0.249**	**2.0**	8.03x
14336	**0.522**	**5.03**	9.63x
57344	**1.44**	19.72	13.69x
114688	**2.79**	39.41	14.12x
229376	**5.5**	78.41	14.26x
458752	10.85	156.08	14.38x

This is not ideal; we see a slowdown factor of 14x in our highly-parallel cases and went from supporting computation of 450 thousand simultaneous commuted waveguides to only 28 thousand. What changed? Two initial ideas come to mind:

Increased branch divergence: Some advice when writing GPU kernels is to avoid *branch divergence* where a portion of the threads in a warp take one path of an `if()` statement but others take the `else()`. This is because GPU threads do not have independent branching logic: the SIMT approach means that all threads execute each instruction in lockstep. In the case of an `if()` statement, threads evaluate the conditional and vote; if they are not unanimous, then both branches are executed in serial and threads ignore execution during the branch they did not take.

Looking at our code, the piece of our code that uses branching is trying to determine whether to loop in a delay line's circular buffer:

```
if (bufferIndex >= waveguideLength) {
  bufferIndex = 0;
}
```

The overhead of running both branches is minimal: we incur an extra instruction of setting a register to zero more often (1 cycle) and the else() branch is a no-op. In addition, GPUs have support for predicated instructions for short branches, which means this case is compiled to the non-branching code:

```
cond = bufferIndex >= waveguideLength
cond? bufferIndex = 0
```

This may be validated by looking at generated PTX pseudoassembly code, or using profiling tools to annotate branch divergence for each line of our source code after a test run.

The second thought of why we see slowdown when introducing heterogeneous delay line lengths is memory access patterns.

Our block of memory for waveguide state was defined to be of size `WARPSIZE*NWAVEGUIDES` by `BUFFERSIZE` rows. This means the N waveguides write to memory locations $0..N-1$ on the first sample, $N..2N-1$ on the second sample, etc.

Global memory access in CUDA is slow, but reads and writes may be coalesced; that is, if all threads in a warp are accessing data in the same aligned 128-byte block, only one to four line reads will need to occur (this is card-dependent). Newer cards have better caches, compiler optimizations, and runtime logic for global memory placement, but this is still worth considering.

In our case, consider we have 32 waveguides in a warp; these are of lengths $5000..5031$. During the first "trip" through the waveguide's circular buffer (first 5000 samples), memory is aligned as all waveguides index to the same offsets. Over the next several cycles through the waveguide, some waveguides will cycle earlier than others and eventually we will reach a state where we require simultaneous memory reads to 32 different lines, so slowdown will result.

Such memory accesses are cached, but with high numbers of waveguides we could easily evict old entries quickly. We open the CUDA Analysis tools, profile memory access, and find that this is indeed the case; Figure 3 shows lots of global memory accesses with only 2% hitting the L1 cache:

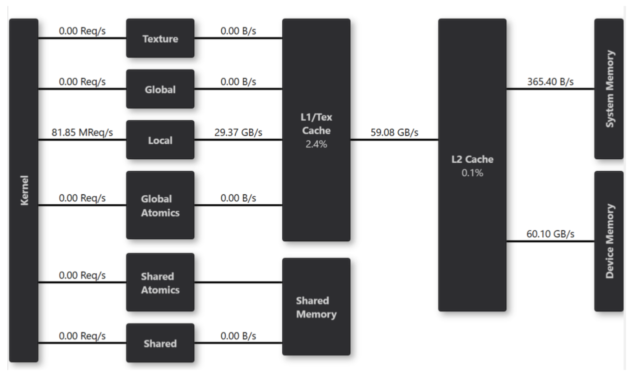

Figure 3: *CUDA memory profiler results. L1 cache hit rate is 2.4%*

To work past this slowdown, we propose two ideas:

3.0.1. Synchronize on Cycle Point

We could determine the longest waveguide in a block and ensure all waveguides' circular buffers loop at the same moment. For example, if we have guides of length 250 and 255, the former avoids writing to memory until our indexing counter loops back to index 1 of the array. The tradeoff is that we need to run more overall cycles in order to completely fill the output buffer from the shorter waveguide[3].

In a degenerate case, what if we have waveguides of length 5000 and 100? Only 2% of cycles are spent actually generating audio for the shorter waveguide with a naïve approach.

We could sort all waveguides by length globally, so that similarly-sized waveguides are in the same warp, to minimize the number of extra iterations–however this makes it much slower to later couple specific waveguides together, which we aim to do in a project that leverages this acceleration.

We could consider a middle ground where we have multiple eligible cycle points. Perhaps every 32 or 64 calculated samples, we could reset and unblock waveguides that are currently idle. This introduces a tradeoff between number of simultaneous memory accesses and number of cycles to compute at the end. On a positive note, in the case of coupling a busy memory controller with the "underpowered" 64-bit Floating Point unit on consumer cards would hide some of the drawback of each.

In the end we did not pursue this approach in depth; a second approach was more promising and produces more readable code:

3.0.2. Shared memory

Shared memory is a type of memory that belongs to a thread block and has much lower latency than global memory, but is only readable by threads within the block that owns it. This sounds great for our current use case and bottleneck. With our card's hardware, we have 96KB of shared memory available to each thread block. This means that we could choose, for example, two warps per block and have 1.5KB of RAM per thread, or 384 32-bit samples per thread.

We may wish to have longer waveguides than 384 samples, so we propose two workarounds:

- Lower utilization: Simply split the 1.5KB up among fewer threads, and use for example 24 out of 32 threads in a warp. While our overall utilization will be lower, the faster memory might save us enough time overall to run multiple copies of our work serially to increase N globally.
- Mixed-size waveguide groups: We could split the available shared memory such that, for example, the two shortest-length waveguides in a warp each donate half their buffer to the longest waveguide.

While the second approach may still seem problematic from a memory access point of view, the rules for shared memory access optimization are different than those for global memory access optimization. Shared memory on our GPU's architecture is grouped into 32 banks, and as long as two threads do not access the same bank at the same time (a *bank conflict*), we obtain full-speed access. On our card, access is actually done in two successive stages, each obtaining results for a half-warp, so the "donation" approach should be safe from slowdown as long as we can arrange memory accesses in time to have no bank conflicts. With simple donation schemes this is straightforward.

[3]we also note writes to that output buffer, previously perfectly aligned, are now unaligned themselves.

As a middle ground, we can sacrifice some compute utilization for larger buffer donations. Consider having each warp provide a developer a predefined "care package" of: (a) One "Extra Large" digital waveguide that occupies 4 banks of shared memory (length configurable, up to 1536 samples of delay), (b) Two "Large" waveguides owning 2 banks each (up to 768 samples each), and (c) 26 normal waveguides (up to 384 samples each).

This still lets us compute 28 waveguides per warp vs. the 32 we had before (87.5% effective utilization) but allows for lower frequency extension. We note that with such approaches we are becoming opinionated concerning the basic system building blocks; when we do this certain applications are enabled but we may block other applications.

We adjust our kernel to use shared memory. Due to the resource configuration of the graphics card and the dimensions of problem, we would use more shared memory than is available in each SM, so we must move from using 64 threads (2 warps) per block down to 32 threads (1 warp), at which point we are under the shared-memory-per-SM limit and our kernel can be scheduled. This serves as a reminder that GPU hardware is not as abstracted as we may be used to when coding for a CPU. In this particular case though, the "nvcc" compiler helpfully caught this at compile-time since it was an oversized static allocation, making for an easy fix.

We also include a quick performance gain of pinning memory on the host, accomplished by simply swapping `malloc` with the API call `cudaMallocHost`.

Results are in Table 5. As before, bolded entries are feasibly realtime. A plot of the same data is in Figure 4.

Table 5: *Time to generate 10 seconds of Audio, "Baseline" uses same-length waveguides, "Differing" experiments use heterogeneous waveguides with either global or shared memory.*

N DWGs	Baseline	differing: global mem...	...shared mem
3584	**0.249**	**2.0**	**0.58**
14336	**0.522**	**5.03**	**0.96**
57344	**1.44**	19.72	**1.82**
114688	**2.79**	39.41	**3.50**
229376	**5.5**	78.41	**6.79**
458752	10.85	156.08	12.59

To summarize: using shared memory allows us to make higher-waveguide counts tractable again. We can still run over a hundred thousand independently-sized waveguides with half of our cycles to spare for extending the algorithm.

At this point we have enough waveguides that we can spend some time thinking of creative applications for them. Those applications will certainly make them computationally more expensive by adding coupling, nonlinearities, modifiable tap points, fractional-length delays, etc.

4. AREAS FOR DEVELOPMENT

We stop here, but note there may still be room acceleration. For example, relatively new GPUs including ours have the ability overlap kernel executions with host/device memory transfers. If we were to double-buffer on the host and device, we can work on one array while the other transfers, and vice versa. This would help us especially at small N or if we wanted to copy hundreds of thousands of individual audio streams back to the host (skipping our current merge step where we sum them per-warp).

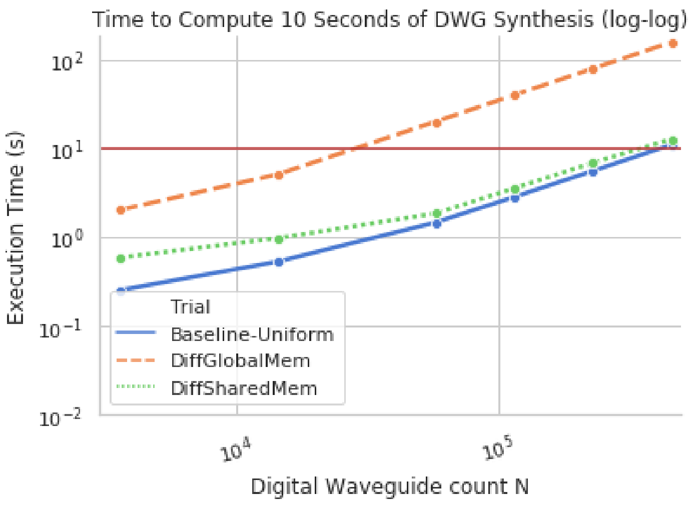

Figure 4: Time to run N heterogeneous waveguides for ten seconds of samples under different trials.

We have been discussing the general-purpose case of supplying waveguides of preconfigured lengths. With a pre-specified configuration, we could write tooling to efficiently group waveguide computations into a warp for maximum resource utilization.

In the modal filter bank, we note that we do not implement phase-correct input re-excitation which is a nice feature supported by these filters: we have logic to track zero-crossings but do not implement parameter updates from the host in a fashion that a "real" system would use. This is a simple and low-cost feature. Furthermore, it is likely that either CPU or GPU should interpolate parameters, which is work that is not being accounted for.

The high performance of these oscillators bodes well if we were to implement a massive collection of digital waveguide oscillators–another case requiring only a few variables and limited multiplies per cycle. It may be worth looking at algorithms that traditionally did well on VLSI architectures for use here, as the concept of many parallel independent instances of a module executing concurrently but varying on input data is shared between the two architectures.

5. CONCLUSIONS

We showed modern consumer GPUs may run high-Q phasor filters and 1D digital waveguides without needing to leverage parallelism across time. In particular, we showed that it is feasible to build a bank of several hundred thousand 1D waveguides, a hundred thousand 64-bit phasor filters with stable per-sample adjustments, or a few million phasor filters at 32-bit precision, all at 44.1kHz.

Reflecting on the the overall optimization development and debugging strategy used here: care must be taken to have the right number of warps grouped together into grids and blocks. Memory accesses should go to the fastest RAM possible, and we need to pay attention to memory alignment. While CPU code does benefit from similar optimizations, GPU algorithms rapidly fall in performance when parameters stray from the ideal range.

One other note around generalizing this code for end users: during development we consulted the capabilities of our particular graphics card several times, to see how many registers we have or to see the various ways we can slice shared memory. While these parameters may be queried from the card at runtime and for the most part newer and more powerful GPUs contain a superset of old resources, this is not a guarantee, and for example if we tried to run our compiled waveguide binary on a GTX 480 from several generations back, it would fail to run because we request too much shared memory.

Still, optimization of these algorithms can be seen as an interesting puzzle; profiling tools make it easy to see where bottlenecks live (if not how to work around them), and it's fun to transpose an array or adjust memory layout and unlock a 10x speedup.

From a sound designer's point of view, being able to use so many of these building blocks at audio rates may allow for higher-fidelity physical models and modal effects, using commodity hardware that often sits idle while working with audio software.

6. ACKNOWLEDGMENTS

Special thanks to conference organizers, and to reviewers for helpful suggestions in improving background and presentation. Thanks to cited authors for providing foundational background for this work: demonstrating feasibility of parallel audio-rate algorithms on the GPU and describing an efficient, numerically stable high-Q filter.

7. REFERENCES

[1] Regina E Collecchia, Miriam A Kolar, and Jonathan S Abel, "A computational acoustic model of the coupled interior architecture of ancient chavín," in *Audio Engineering Society Convention 133*. Audio Engineering Society, 2012.

[2] Jonathan S. Abel, Sean Coffin, and Kyle Spratt, "A modal architecture for artificial reverberation with application to room acoustics modeling," in *Audio Engineering Society Convention 137*, Oct 2014.

[3] Julius O. Smith III, "Physical modeling using digital waveguides," *Computer music journal*, vol. 16, no. 4, pp. 74–91, 1992.

[4] Julius O Smith III and Scott A Van Duyne, "Commuted piano synthesis.," in *ICMC*, 1995.

[5] Lauri Savioja, Vesa Välimäki, and Julius O Smith, "Audio signal processing using graphics processing units," *Journal of the Audio Engineering Society*, vol. 59, no. 1/2, pp. 3–19, 2011.

[6] Lauri Savioja, Vesa Välimäki, and Julius O. Smith III, "Real-time additive synthesis with one million sinusoids using a gpu," *128th Audio Engineering Society Convention 2010*, vol. 1, 05 2010.

[7] Fernando Trebien and Manuel Oliveira, "Realistic real-time sound re-synthesis and processing for interactive virtual worlds," *The Visual Computer*, vol. 25, pp. 469–477, 05 2009.

[8] Jose Belloch, Balazs Bank, Lauri Savioja, Alberto Gonzalez, and Vesa Välimäki, "Multi-channel iir filtering of audio signals using a gpu," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 05 2014, pp. 6692–6696.

[9] Belloch Rodríguez and José Antonio, *Performance Improvement of Multichannel Audio by Graphics Processing Units*, Ph.D. thesis, 2014.

[10] 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, 2017.

[11] 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.

[12] Max Mathews and Julius O. Smith III, "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)*, 2003.

[13] NVIDIA Corporation, "NVIDIA CUDA toolkit documentation," `https://docs.nvidia.com/cuda/`, [Online].