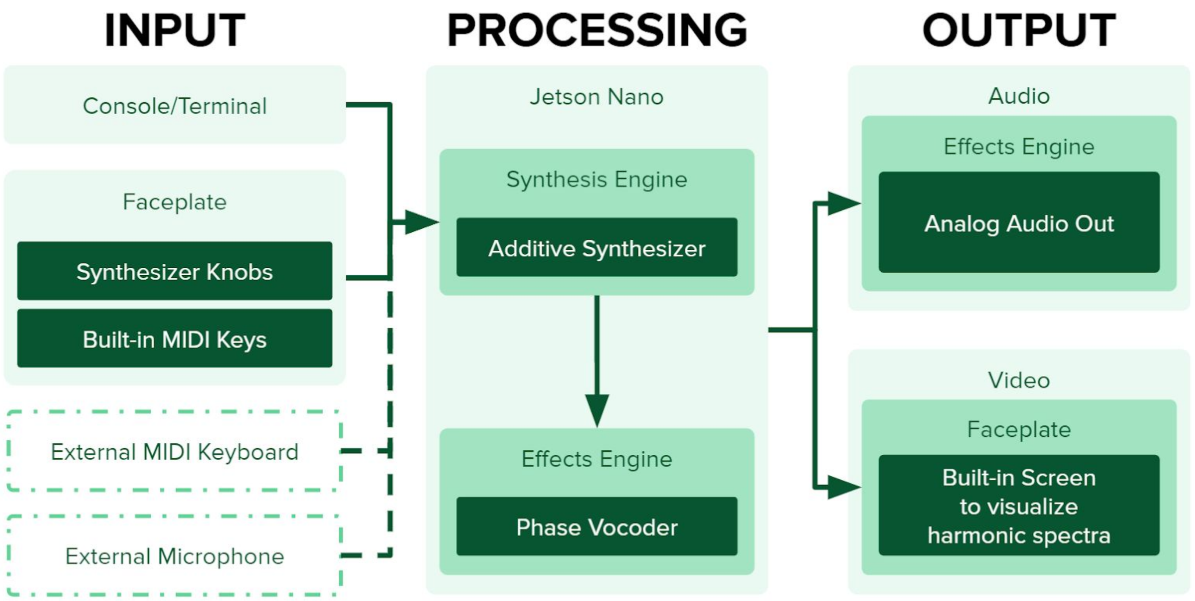


Alternative Audio Processing

Team #17

Davis Polito, CMPE, dpolito@seas.upenn.edu
Nikil Ragav, EE M&T, nikilr@seas.upenn.edu
Mason Elms, MEAM, melms@seas.upenn.edu
Enoch Solano, CIS, enochs@seas.upenn.edu

Advisor:
Tania Khanna, taniak@seas.upenn.edu

I. Table of Contents

II. Executive Summary

The current state of music production equipment is riddled with barriers to entry for aspiring producers. Powerful hardware synthesizer equipment and digital audio workstations (DAWs) are both prohibitively expensive. DAWs provide a full production suite with all necessary tools, but lack intuitive physical control. Hardware synthesizers are tactile but provide only narrow functionality, meaning one cannot produce a full song on a single hardware synthesizer.

Our team set out to develop an inexpensive, portable, and powerful hardware synthesizer to democratize access to music production tools, as well as to provide established producers with more effective equipment for specific applications.

We began with the goal of shrinking the power of software into a portable package by parallelizing audio computations. We set out to develop applications taking advantage of a Graphics Processing Unit (GPU)'s natural parallelization ability, using the Nvidia Jetson Nano as the processing engine. Before the outbreak of COVID-19, we had planned to outsource manufacturing of a PCB and heatsink to external vendors given the complexity of those components. In addition to this, we planned to develop a GUI using a Qt framework that would produce the same signals as would our physical controls. We would thus be able to develop the synth engine to interface with this GUI and in theory, the synth engine would be able to interface with the physical controls once ready. In order to get the physical controls ready, our plan was to order our designed components and assemble the hardware product in time for the final demo. Obviously we were unable to follow through with this.

Our team decided that we wanted to continue developing our project after the transition to online courses. To accomplish this, we pivoted to a software implementation of our ideas. We used JUCE, a set of libraries made by music hardware and software company Roli, to create a GUI that would mimic the intended design of our original product. We continued to develop and debug the synthesis engine and phase vocoder until the day of our demo. Ultimately, we were successful in creating an audio application that accomplished our intended goals on the Jetson Nano, although it remained buggy given the time frame.

III. Overview of Project

In our group, 3 of us are active musicians, and 2 of us produce electronic music. We are constantly uninspired and frustrated by music production software. Our brains come up with music faster than our computer mice can drag a virtual knob. Things that should be as simple as turning a knob or humming a tune take 10 times longer than they should. The unintuitive software interfaces cost $500 for a license. Hardware synthesizers and electronic instruments are tactile and more intuitive to use, but they are not portable. Additionally, to make a full song, you'd need upwards of $1200 of equipment.

Our product combines the power of software with the intuitive control of hardware into a handheld device the size of a Nintendo Switch. We accomplish this by using the same Graphics Card in the Nintendo Switch, but instead of rendering video, we process audio in parallel. Just like the Nintendo Switch has swappable controllers, we have a swappable faceplate so that depending on the use case (playing a keyboard, singing live on stage, using a guitar foot pedal), a musician can use the same hardware with tailored UI.

The hardware synthesizer market has grown rapidly in the past 4 years. Based on rough estimates of market size of hardware and software synthesizers, we believe we can sell around 50,000-300,000 units of our device. At that scale, because we are using 80% of the same components as the Switch, we think we can hit a price target of around $400. Because we are producing effects with software, and because our hardware box has interchangeable faceplates, we could potentially sell the same box to not just music producers but also to live singers, drummers, and guitarists, potentially tripling or quadrupling the number of units we could sell. This scale could reduce our marginal cost significantly.

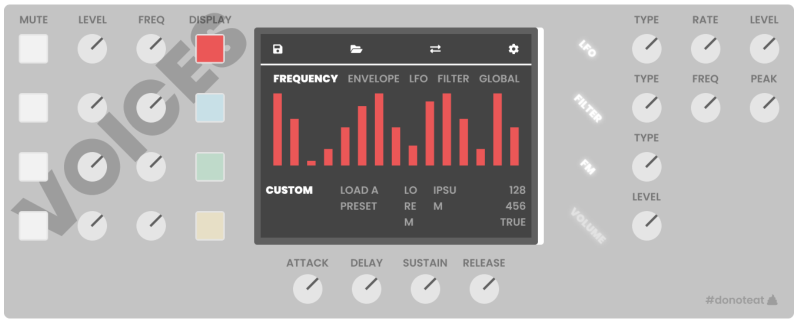
IV. Value Proposition and Stakeholders

Our product makes the process of producing music more intuitive and tactile. This makes it easier for “bedroom producers” (hobbyists and novices) to pick up and enables experienced indie and studio producers to make unique musical sounds faster.

Because we are using the emerging technology of a GPU to produce the music, we are able to produce musical effects such as autotune in real-time. It is currently not possible to autotune for live performances with computer software unless a performer has several desktops chained together.

We have confirmed our value proposition by interviewing several prototypical customers and influencers during our time in the NSF Innovation Corps program. We have met with several youtube electronic musicians and influencers that have more than 300,000 followers. They have expressed interest in using our first beta devices.

Our Hardware Synthesizer device consists of 1) a "brain" with the GPU and audio inputs/outputs, 2) an interchangeable control faceplate, and 3) audio software engines. This is similar to the Nintendo Switch, which includes the console, interchangeable controllers, and video games.



Our senior-design prototype faceplate design

V. Market Opportunity and Customer Segments

Our modular design allows us to design affordable customized faceplates for specific use-cases/segments and also achieve economies of scale on the most expensive "brain" component.

For the additive synthesizer faceplate, our initial segments are "bedroom producers" of electronic music and indie professional influencers. For real-time autotune and harmonization, our initial target market includes live performers.

In the future, we plan to build other control faceplates such a foot pedal so that guitarists can also use our product for special wowow and reverb effects. We could also partner with brands like Moog to build a faceplate that feels like their retro products and simulate their sounds using our GPU.

The global digital music software market is estimated at $200M. It is expected to grow at 3.2% CAGR as per NAMM. The global digital instruments market is estimated at $4.5B, and this market includes synthesizers, keyboards, and guitars. This is growing by 4.4% CAGR according to Statista.

VI. Competition

There are very few popular software synthesizer DAW softwares. These include Logic Pro (Apple), Ableton, and FL Studio. There are many more developers of VST plugins which are used to produce custom sounds, reverb effects, etc. The most popular hardware synthesizers come from companies like Roland and Korg. The synthesizer space is growing rapidly with upstarts like MakeNoise. However, only Teenage Engineering is in the portable synthesizer space with a low-volume $1200 and $500 product with cult followings.

Based on sales of comparable products and software, we think we can sell around 50,000-300,000 units.

We believe our competitive advantages include our business model as well as our technology innovation. Programming a GPU is not something that software and firmware developers common to the music industry can do easily - it requires specialized knowledge, and the process is almost orthogonal.

VII. Revenue Model

Our revenue model is inspired by the video game industry and designed to democratize music production. We reduce barriers to entry for novices to produce full songs, and for experts, we sell add-ons.

Because our “brain” shares almost 80% of components with the Nintendo Switch, at a volume of 50,000 units, we believe we can hit a price target of $300-400. From there, additional faceplates could be sold just like controllers/games for $50-100. We also plan to have an app-store for new sounds and software modules.

Worst case, if the hardware fails, because many computers use the same NVIDIA graphics cards, we could turn around the software we've already developed and sell a VST plugin for existing DAW softwares. At the end of the semester, due to COVID, we developed such a software using the JUCE library.

VIII. Cost

Our biggest costs include the costs of mass-manufacturing. After proving the market with a small batch (10-100) of expensive beta units, we hope to join the HAX accelerator and find 5x cheaper mass-manufacturing in China. We plan to drop ship directly from our own website. We will build our market organically through the influencers who offered to beta test and feature our product in their YouTube videos.

IX. Technical Description

Our constraints and customer expectations informed our specs.

Realistically, as a senior design project, we did not expect to design the full manufacturable product by the end of the second semester (although we would love as much as guidance and advice to get there!)

We simply do not have the experience or a checklist needed to pre-emptively avoid design issues, multiple prototypes, and engineering validation builds. We also lack the proper equipment to test the high-frequency multi-gigahertz, impedance-controlled lines between the graphics card, RAM, and Superspeed USB.

We are capable, however, of designing a fully-manufacture-ready faceplate, as it has no major high speed components. The fastest speeds are 60MHz for USB 2.0, so we have the equipment to test, and tolerances don't have to be as precise. We didn't get to complete this as COVID disrupted PCB fabs. As such, we developed the fully-software version of the synthesizer.

The following specs reflect the hardware we would develop in the future as/when we continue the project.

We interviewed 20 music producers to understand what people are looking for in a portable music synthesizer. This list of music producers included YouTube musicians with over 300K followers who regularly use a portable synthesizer similar to our intended product, but 3 times the cost.

Our music producers all mentioned that they want to be able to start making music as quickly as possible, specifically they should be able to play some sound within 30 seconds of turning the device on / plugging it in. This requirement means we will eventually have to package our own custom build of linux to turn off unnecessary features.

Our target form factor was roughly the same as the Nintendo Switch or Teenage Engineering OP1, at around 10 inches wide, 4 inches tall, and 15mm thick. Achieving such a thin form factor with proper cooling of electronic components is extremely difficult unless we implement vapor chamber cooling, which would greatly increase the cost and complexity. However, after talking to producers, they felt that the initial version simply needs to be small and light enough to fit in a backpack, so anything smaller and lighter than a laptop is fine. As such, we have relaxed our initial binder board dimensions for our faceplate to 13.5 inches by 5.5 inches. With the stock heatsink, the device would measure approximately 45mm thick, including the faceplate.

We originally envisioned that music producers would want to be able to pull out our synthesiser from their bag and make music on a train, on a plane, or at a park. As such it needed to have a battery that would last for at least 10 hours. However, producers said that there's no point of a battery if the device is not capable of producing full songs completely on its own.

We will not have a battery for the first prototype board. We will eventually include sequencer software which will allow users to save full songs directly on the device. Because there are still multiple uses such as on-stage or as a foot pedal, we will still include a battery.

Based on benchmarks of the Nvidia Jetson Nano and of the Nintendo Switch, we think we can get the device to run at around 3 W average draw while the GPU is processing sounds with 12 W peak draw. Idle draw will be less than a Watt. The device approximately will spend 50% of its time at half a Watt simply reading input from the faceplate and displaying information to the screen, 40% of its time at 3W processing and outputting sound waves Because we are targeting ~10 hrs battery life, we will need roughly (.5 * .5 W + .4 * 3 W + .1 * 12 W) * 10 hrs = 26.5 Wh battery capacity. With a closed cell potential of around 3.7 V for a lipoly battery, we will need approximately 7000 mAh.

We need our box to use the industry standard MIDI protocol because, based on our conversations with producers, the option to be able to plug in their favorite keyboard or drumpad into our device is vital to their interest. Additionally, MIDI support means that each of our faceplates can function as MIDI controllers. As every device, DAW, and producer uses MIDI interfaces to play and control music, it is imperative that our device do so as well if it is to fit in the current market.

Our most important specification is affordability. Price changes with the components used, and it also changes drastically with scale (number of units produced). We want to hit a price target of $400 so that our product will be cheaper than a software license when we eventually release our final product. This means that the production cost should be closer to $200 per device at scale.

This final product will be the "brain" box containing the GPU, ports, ADC, and DAC and one faceplate that slides onto the brain.

Our prototype setup can be had for just under $400. The NVIDIA Jetson Nano for $100, the HiFi Berry ADC and DAC at $70, WiFi+Bluetooth card at $20, custom PCB for the faceplate at roughly $135 given the size and components, and $30 for materials for the casing.

Regarding functionality, we were presented with effectively a blank slate.

The goals were to design functions that would take advantage of parallelization on the GPU, so we began by considering existing processes that we could improve by using the Nano.

Additive synthesis is parallelizable because it represents a complex wave as a sum of sinusoids. The oscillation of each individual sine wave can be computed at an individual timestep without care for the other waves in that step allowing for parallelization.

When considering effects, we decided upon phase vocoding because of its accessibility in addition to its parallelizable nature. Processes such as autotune and harmonization are historically too complex to run on a typical microcontroller. Thus in order to use them live artists must run a computer in the background. By running the brunt of the computational load in parallel we hope to port these effects to the jetson nano. Our goal in doing so is to provide smaller artists without the money for real infrastructure, or who may be performing without a full gig setup, with the opportunity to use autotune and harmonization live in the same way they would use a stomp-box or effects box.

Our product definition has been through several iterations based on both customer feedback and technical feasibility.

Initially, our intended form factor was closer to that of a Nintendo Switch. After customer feedback, we determined that the small size was less essential for our initial prototypes since we will not have software ready to produce a full song. This frees up more space for our controls and makes our PCB layout easier.

Additionally, we toyed with the concept of real-time modelling of sound waves or instruments. For example, we were hoping to model the shockwaves that result from hitting a drum or cymbal so that we could create much more realistic digital instruments. After looking into a paper discussing the 3D modelling of flat plates, talking with the Stanford research group behind it, and delving into the nitty-gritty of acoustic modelling, we determined that this was unlikely to be accomplished with small enough latency to be usable for musicians.

Detailed description of each component and engineering process

Additive Synthesizer

An additive synthesizer generates sounds by adding pure sine waves of varying frequencies and amplitudes, and playing back the summation of said sine waves.

The computational cost in building an additive synthesizer is in the computation of sinusoids and specifically the quantity of said sinusoids. We get around this issue by computing the desired sinusoids in parallel within the GPU and summing them in parallel as well.
We then operate on the values in the CPU as desired. This is effectively the backend of the synthesizer.

The frontend of the synthesizer allows the user to interact with voices among other effects like ADSR. Voices are the complex waves composed of simple harmonics (i.e. pure sinusoids). A user interacts with these voices by either loading in preset voices, modifying the amplitude of the entire voice or specifying individual amplitudes/frequencies of each harmonic.

This interaction is partially through (virtual) knobs and buttons, and partially through a graphical window. Once the voices and other effects are specified, a user is allowed the play voices by simply pressing down on keys.

Additive Synthesizer (ADSR)

Instead of just playing back the pure summation of harmonics, the summation amplitude is scaled according to an ADSR (attack, delay, sustain, release) envelope. In our synth engine, each voice has its own ADSR envelope.

The user is able to select a voice and modify the ADSR that envelopes said voice. The user is able to specify the attack parameter, the decay parameter, the sustain parameter, and the release parameter through the knobs provided on the GUI/physical controls. The attack parameter specifies the time it takes for the voice to reach the max amplitude, the decay parameter specifies the amount of time it takes to reach the amplitude specified by the sustain parameter, the sustain parameter specifies the amplitude maintained while a user holds down a key, and the release parameter specifies the amount of time it takes to reach a zero amplitude once the user releases a key.

Additive Synthesizer (LFO)

Another effect that the additive synthesizer offered was a low frequency oscillator (LFO) that could be applied on amplitude of the summation of all of the voices. Through the controls given to the user, a user was able to specify the frequency of the LFO, the amplitude of the LFO, and the type of LFO (e.g. sine wave, square wave). Once specified, an LFO was applied to summation of voices by adding some offset to the amplitude of the summation of all of the voices.

Additive Synthesizer (JUCE Implementation)

The JUCE Framework allowed for easier implementation of audio in and out through its boilerplate process block coding methodology. Setting up JUCE for Jetson Nano required an intense set of build system workarounds. We first had to compile the system to work for Jetson Nano's Arm64 architecture, which required going into the Linux makefile for Projucer (JUCE's application initialisation program) and editing the compilation variable. Once the program was built, we used another open source project called FRUT (haha you get it?) to convert the "*.jucer" project into a program that compiled via CMake. CMake is an incredibly powerful build environment allowing integrated compilation of different programming languages and architectures into one executable program. This CMake file was in effect a wrapper or an interface to the jucer program meaning that some care was required in adding new libraries and functions greatly increasing the time needed to work with it. We compiled our CUDA Additive

Synthesis Engine as a separate library, and were eventually able to include it in the JUCE program. The additive synthesis engine did not change from the original implementation.

Implementing the control in JUCE required a great refactoring of the code. Since JUCE introduced the GUI that we have talked about as a separate module rather than parameters that we were read into, we needed to add many functions that would reveal our internal state in an object oriented manner. For example instead of adding individual sine waves via a frequency and gain, we initialize each voice to have a set of *N* harmonics whenever the fundamental frequency changes. These harmonics are all set to a gain of 0 unless otherwise changed by the system. This helps to reduce the number of unnecessary array accesses per block production.

The block based processing of audio in JUCE is the same as in the RTAudio engine we had previously used, except that it allows for simpler and more immediate manipulation of the output blocks. This allowed for the final ADSR to be a much simpler built-in function rather than the custom per voice ADSRs.

Additive Synthesizer (Qt GUI)

Qt is an application development framework for desktop, embedded and mobile. It's not specifically tailored to an audio production setting, but is instead a general widget toolkit. We developed this before the lockdown in order to produce the same output that we were planning on having the physical controls. Specifically, the Qt GUI developed outputs a JSON packet that contains the state of the board (e.g. which buttons are pressed, the absolute positions of the knobs) and this JSON packet is produced every few milliseconds or so. This allowed us to develop an interface for the synth engine that read in a JSON packet and interpreted the state of the board to commands that could be sent to the synth engine (e.g. update the type of LFO). In theory, our physical controls would just need to output the same type of JSON packet and the synth engine would work with said controls right away.

Phase Vocoder

Phase Vocoding consists of three stages, analysis, processing, and resynthesis. Analysis of the input signal obtains both the continuous and discrete phase information of the signal through a Short Time Fourier Transform or windowed Fourier Transform. The continuous phase information can be obtained using the windowing operation, since the power of the signal is shared across multiple Fourier Transform windows. In the processing phase, time stretching or pitch shifting occurs. Time stretching occurs through multiplication of phase and windowing hopsize in order to spread the signal across a larger portion of the output. On resynthesis the phase information is converted back to its time components through an inverse Fourier transform or additive synthesis.

The process for real time action of the above algorithm moves from all the stages happening at individual stages on the signal to streams spawning once each CPU read occurs.

The CPU buffers in *N* samples of the input signal at a time. When a collection *N* samples is received (referred to as a frame) the GPU spawns N/hopsize streams to process the audio. The stages occur the same as they would in offline synthesis save for this portion of spawning portions

Faceplate Circuit and Embedded Firmware

The faceplate is the circuit board that contains all of the control knobs and buttons. It connects to the main Jetson Nano board (and eventually our custom board) via USB protocol.

The embedded microcontroller on the board uses a timer-interrupt based scheduler to scan the position of every knob and the state of every button every 0.1 ms. Every 0.5 ms, the microcontroller compiles all of the states into a single message and sends the message to the main board via USB.

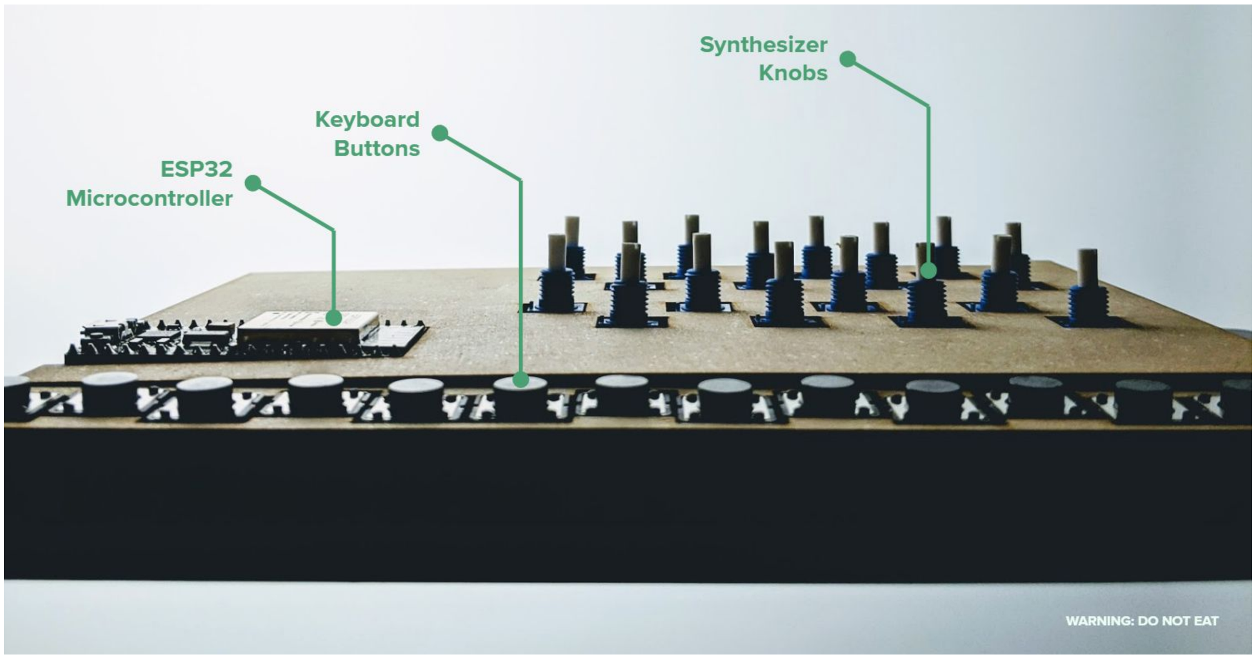


Figure 1: Protoboard version of the faceplate

The protoboard faceplate (faceplate v1) from the first semester used an ESP32 microcontroller. The final version (faceplate v3) will utilize an STMF4 so that we don't have to use an FTDI Serial to USB converter chip and so that debugging and DFU (firmware update) can be easier. We looked into sending the messages via DMA (direct memory access over UART) to minimize latency, but since the device is anyway using USB to connect to third-party accessory keyboards, and the latency is already better than the noticeable threshold of 1 ms (in other words, this communication is not the bottleneck for users' perceived latency), we are planning to just use USB. The added benefit of using USB protocol is that with almost no additional engineering we could make the same faceplate into a controller that plugs into a computer in case we wanted to sell the control unit standalone without the synthesizer brain.

With the protoboard version (v1) of the faceplate, the knobs are simple potentiometers, but in the final version (v3), they will be magnetic encoders that use a Hall-effect sensor to detect the position of a rotating magnet. The encoders then digitally output the current relative or absolute position (depending on the sensor model used). This also can allow for detecting a button press.

We very early on ran into an issue where we did not have enough GPIO pins on the microcontroller to connect all of the peripherals. To save pins, we had to get a little creative by using SPI knob encoders and using a button matrix instead of connecting each button to a dedicated GPIO pin. The button matrix means that we can just use 8 pins for 16 buttons instead of 16 pins. We use the timer interrupt to sequentially toggle one “column” high and then read the value at each row to see which buttons have been pressed.

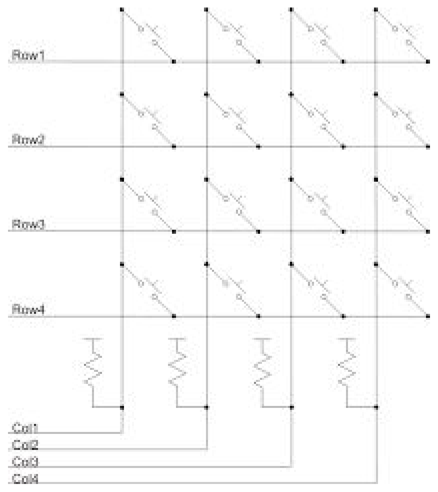
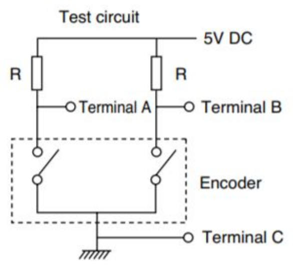


Figure 2: Circuit diagram of prototype faceplate buttons

In the second semester, we started developing the schematic for the first PCB version of the faceplate (faceplate v2). We realized that the I2C magnetic sensors that we wanted to use for all of the knobs and buttons made the knobs too difficult to assemble (this is because the part that the user touches does not make any physical contact with the sensor on the PCB, so the knob needs to be mounted directly to the casing with a ball-bearing on some kind of butterfly switch. These components require too-high precision to make ourselves, and we struggled to find a place that could make this assembly for us). Thus we ended up reverting to mechanical quadrature rotary encoders.



The problem with these knobs is that they are essentially 3 buttons, so we again returned to not having enough GPIO pins.

We considered multiplexing and charlie-plexing, but at the end we decided that it was not worth writing complicated firmware to constantly scan and record the values of each knob.

After reading a Nintendo Switch controller reverse engineering blog, which mentioned that their bluetooth controller had a built in keypad scanner, we realized we could use a keypad scanner to achieve our desired result and found such a component that we could use. It connects to the microcontroller with I2C, so we only need 2 wires and much simpler software to scan the status of every button and knob. Given the number of buttons and knobs we have, we planned to use 3 of these scanners.

From the recommendation of Eduardo Garcia from Bresslergroup, we were planning to use STM’s TouchGFX library for the touchscreen interface. We found a controller and SPI touchscreen that we could use from their recommended list and also sat in their workshop presentation. By the time we got to the point where we were about to order our components, the touchscreen went out of stock and alternatives with a similar size and resolution that could be controlled via SPI could not be found. Their devboard with the screen and microcontroller went from \$100 to \$500. The microcontroller alone went from \$5 to \$15. Additionally, we felt that there wasn’t much point to code a second version of the UI using a totally different platform and language because it would slow down our ability to iterate and fix bugs. We thus decoupled the screen from the controller board and instead used an HDMI touch screen directly connected to the Jetson Nano.

For our LED buttons that change color on-press, we planned to use WS2812 RGB LEDs which are controlled via I2C. This would have been similar to the Adafruit Neotrellis boards. By the time we got to implementing this in our schematic, the whole factory systems had shut down due to COVID.

We had developed the board outline, board stack up (the specifications for each layer of the PCB) and the assembly of the entire device.

After we eventually get a chance to build and debug our v2 board, we will update our plans for the final faceplate (v3). For now in v3, we are planning to use the same components on the HDMI touch screen that we procured and incorporate that into our faceplate board. If we have access to testing for gigahertz-speed signals, we could implement a USB C 3.2 connector on our faceplate board such that the single connector carries both HDMI and USB for the touchscreen and MIDI over USB for the knobs/buttons.

Thermal Simulation

The simulation of heat sinks is, fortunately, well-understood and documented online. The boundary conditions were selected as follows.

Convection at the tips and bases, as per real conditions. Values were estimated from research to be 50 W/m2 for the tips and 6 w/m2 for the base. The heat generated by the device was represented as a 10W heat flux on a square portion matching the size of the GPU chip. Maximum power draw is stated online to be 10W, and the simulation assumes the worst case (that all 10W are dissipated as heat through the chip). Conduction through the geometry was defined by the material conditions. Aluminium 6061 was used as it is a common alloy used in manufacturing, and there is no information about the specific alloy used. A simple heat flux was used instead of radiation, again due to license limitations, estimated to be 3W in the area of the heatsink that is raised to prevent the CPU inductor, GPU inductor, and PMIC from crashing with the base.

The geometry of the heatsink itself was taken from an online CAD model of the Jetson Nano provided by NVIDIA. Once opened in solidworks, dimensions were checked against the physical device to ensure real-world accuracy. The .SLDPRT file was exported as a CREO parametric file, to avoid compatibility issues and retain the full shape, and edited in the ANSYS SpaceClaim editor to merge several curved faces that were rendered as multiple

The mesh was defined based around license limitations. The maximum base element size was 2e-4m, with a refinement on the tip geometries (any places where the geometry was smaller than 2e-4) at a size of 7.5e-5m. This was the smallest size possible given the license restrictions on elements and nodes, 32k total for mechanical calculations.Due to the same license restrictions, a shortcut had to be taken to mesh the entire geometry of the heatsink. The geometry was cut in half and a symmetry condition was imposed on the boundary.

Transient analysis was chosen for ease of comparison to physical testing. Results from both built in sensors and contact heat sensors will provide data over time, and being able to compare the simulation as it runs over time will allow for more accurate validation of results. Once the final case design has been settled, if the factory heatsink is too large, the validated simulation setup will be used with parametrically controlled geometries - pin fin, square fin, different shapes depending on available heatsink options and PML manufacturing abilities - to determine the most effective design given space constraints.

The simulation results were validated through stress testing of the Jetson using GLMark2. While GLMark2 is described as relatively uninformative for modern computers, the power and size of the Jetson Nano makes it a viable testing tool. The standard benchmark loop for GLMark2 was run on a 29 inch, 21:9 monitor for 45 minutes. Data was recorded using a multimeter with a thermal sensor attachment, as well as with the Nano's internal temperature log.

GUI and JUCE

As mentioned, we pivoted to using Roli's JUCE libraries to create and implement an interface that would be able to enact real time state changes in our engine. The goals were to allow for control of 32 harmonics in each of four voices, each with their own adjustable ADSR envelope. In addition, we implemented a variable waveform LFO and filter. Given the time constraints, these were accomplished to varying degrees of success.

The engine code (Appendix A) remained largely unchanged from what we would have used for a physical implementation. As such, it was compiled using CMAKE alongside our other files. The JUCE library makes use of custom component classes, with a variety of hierarchies and inheritances. The overall application structure runs from a 'Main.cpp' file, which instantiates a 'MainComponent' from 'MainComponent.cpp.' This houses the audio signal blocks and buffer, as well as the GUI window component. The GUI window holds all GUI objects (knobs, sliders, buttons, etc) as its children, as well as a subcomponent called 'Display' that handles the harmonic control of each voice. A single GUI component and an engine are instantiated on startup.

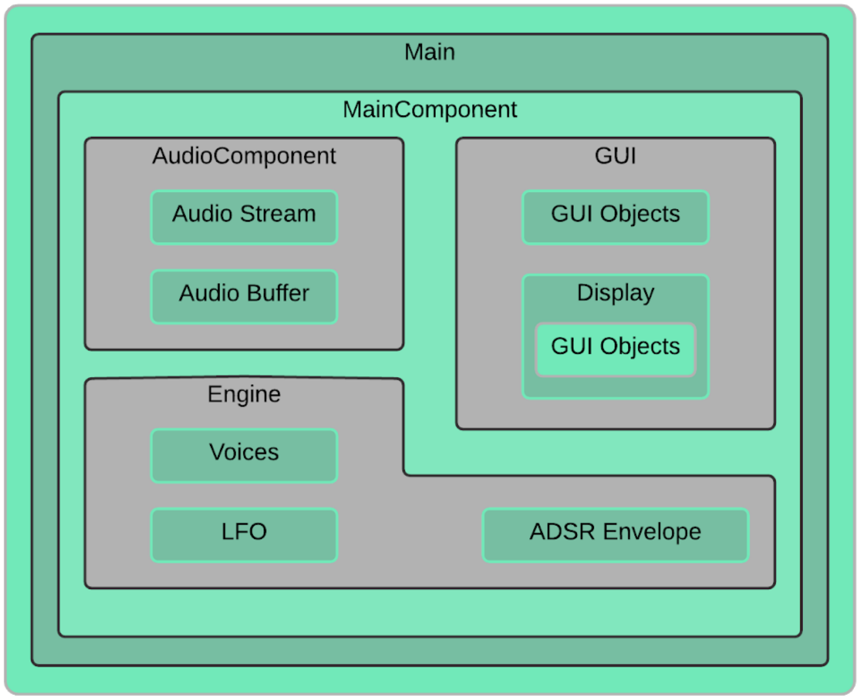


Figure (): Rough diagram of JUCE implementation hierarchy

To begin the GUI implementation, the JUCE graphical GUI editor was used to create the basic structure. As development progressed, it became apparent that the level of control offered in the editor was not sufficient. The existing code was copied to a new file and built upon to allow for the changing of button colors to indicate mute on/off and voice selection. The Display file was created separately to allow for easier handling of switching the voice displays. The entirety of the GUI was defined using pointers which were assigned during construction. This allowed for reduced functional overhead in the memory-constrained Jetson Nano.

Final Status

Our Additive Synthesizer is currently operating at peak efficiency based on the paper written by Professor Savioja. Additive Synthesis on a CPU is very straightforward. Computation occurs in a nested for loop where the inner loop computes the sum of all the sine waves at time *t*(represented as an angle in radians). This can be naively parallelized by expanding that outer for loop into threads on a GPU kernel. This is done in the simple implementation with relative success.

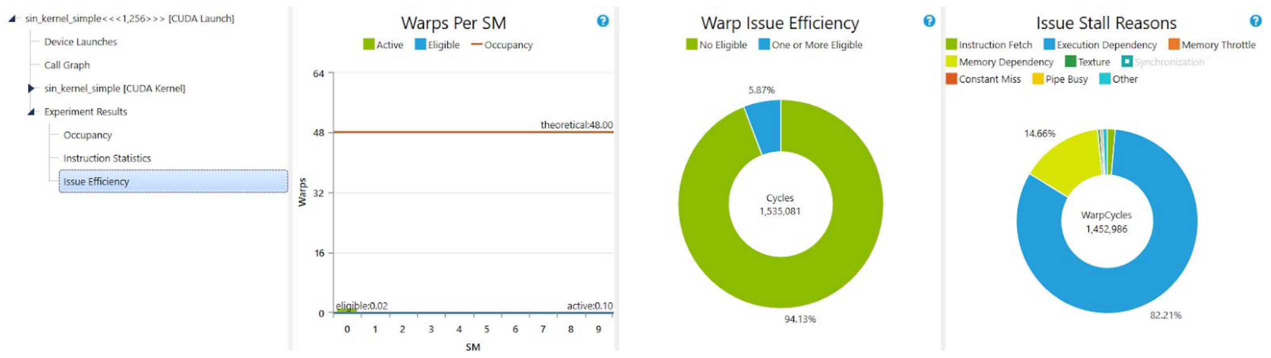


Figure 3: The diagram illustrates the inefficiencies of the simple kernel algorithm, which simply sums all of the sinusoids of a sample in a for-loop in every thread. This results in frequent execution stalls as can be seen (depicted as blue) in the second pie chart.

As you can see, there is little utilization of the GPU and most of the time is spent waiting for process execution with no warps that can execute. This is because looping in a GPU kernel is never suggested. We would prefer to compute each individual sine wave at time *t* in separate kernels as well as spread the computation of those samples across various blocks. We then sum those blocks individually.

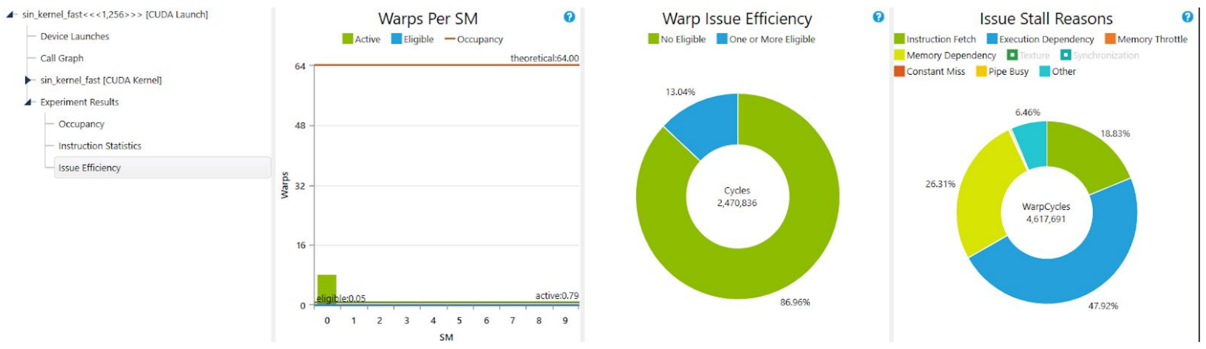
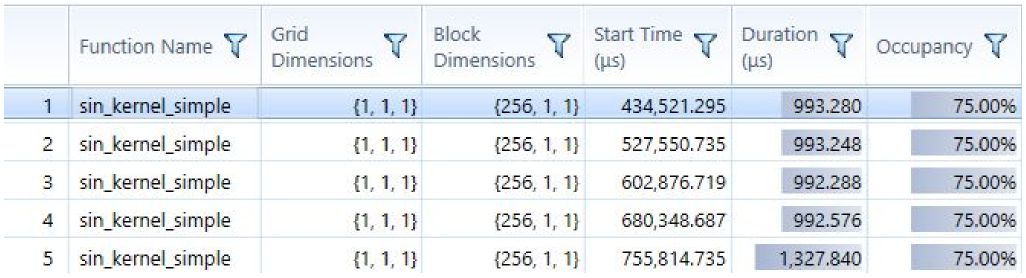


Figure 4: The diagram illustrates the improvement of the fast kernel algorithm, which splits up each for-loop into multiple threads. This reduces the frequency of execution stalls.

This may be a more complicated explanation than the average person can understand so we present to you the execution speed data to better explain improvements.

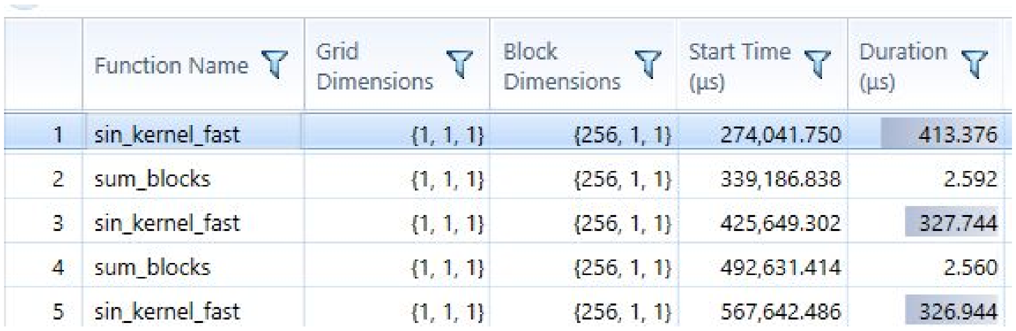
The simple kernel has an average execution time of ~1 millisecond:



	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy
1	sin_kernel_simple	{1, 1, 1}	{256, 1, 1}	434,521.295	993.280	75.00%
2	sin_kernel_simple	{1, 1, 1}	{256, 1, 1}	527,550.735	993.248	75.00%
3	sin_kernel_simple	{1, 1, 1}	{256, 1, 1}	602,876.719	992.288	75.00%
4	sin_kernel_simple	{1, 1, 1}	{256, 1, 1}	680,348.687	992.576	75.00%
5	sin_kernel_simple	{1, 1, 1}	{256, 1, 1}	755,814.735	1,327.840	75.00%

Figure 5: Average duration of executing the simple kernel algorithm.

Whereas the fast kernel has an average execution time of less than half a millisecond:



	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)
1	sin_kernel_fast	{1, 1, 1}	{256, 1, 1}	274,041.750	413.376
2	sum_blocks	{1, 1, 1}	{256, 1, 1}	339,186.838	2.592
3	sin_kernel_fast	{1, 1, 1}	{256, 1, 1}	425,649.302	327.744
4	sum_blocks	{1, 1, 1}	{256, 1, 1}	492,631.414	2.560
5	sin_kernel_fast	{1, 1, 1}	{256, 1, 1}	567,642.486	326.944

FIgure 6: Average duration of executing the fast kernel algorithm.

Our second feature, the Phase Vocoding algorithm, does not currently run in real time due to complications from the algorithm. Our preliminary timing/testing shows that we should reasonably be able to implement the real time algorithm. From the figure below it can be seen that at most our program takes about 1.9 ms to complete. Since we plan to run at less than 256 samples per frame in order to reduce latency (256 samples corresponds to ~2.9ms at a sampling rate of 44.1khz)

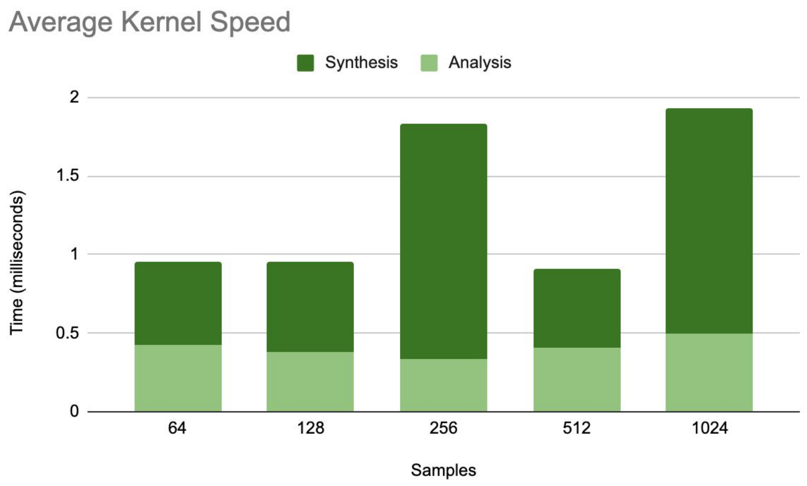


Figure 7: The average time it takes the Phase Vocoding algorithm (split into synthesis and analysis) to compute the corresponding number of samples.

The process of running the algorithm in real time is simple. Whenever a buffer of samples is received on the CPU side the GPU will spawn N/hopsize streams in order to process the audio. Below is a visual to explain the process

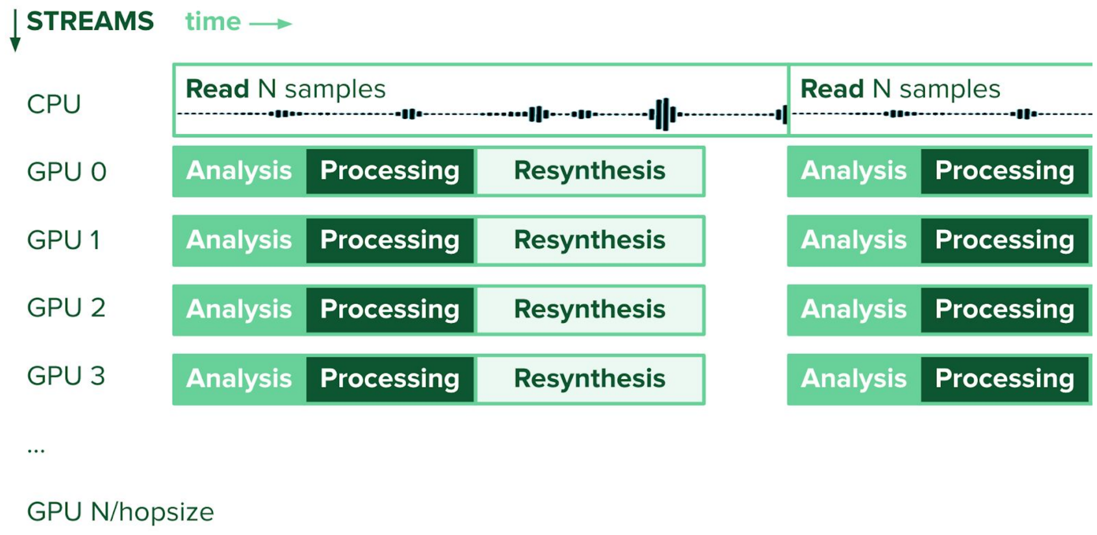


Figure 8: Pictorial explanation of how the GPU analyzes, processes, and resynthesizes the samples in parallel.

A thermal model has been implemented in ANSYS Student 19.1 using the Mechanical solver suite. Thermal power output was set to 10W, given some online forum postings and performance benchmarks by reviewers. The simulation was run over a ten minute span to ensure a steady state was reached. These results will be validated through physical benchmarking in the next semester.

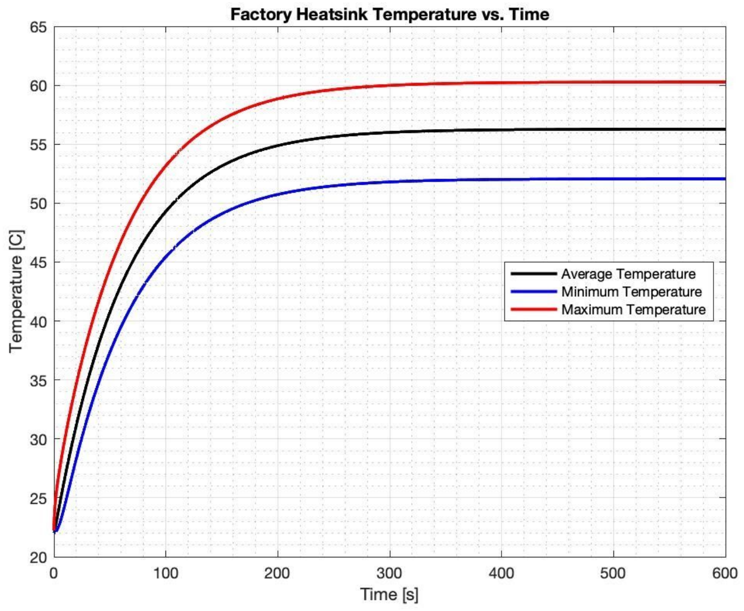


Figure 9: Average, Maximum, and Minimum Temperature over the 600 seconds of simulation

As shown in the plot of the thermal data, the temperatures reached equilibrium at around 5 minutes of testing, with T_avg sitting at 57.274 [C]. The maximum reached a value of 60.2690 [C], well below the stated throttling speed of the integral components, 90[C]. As expected, the maximum temperature occurred at the location of the processing chip.

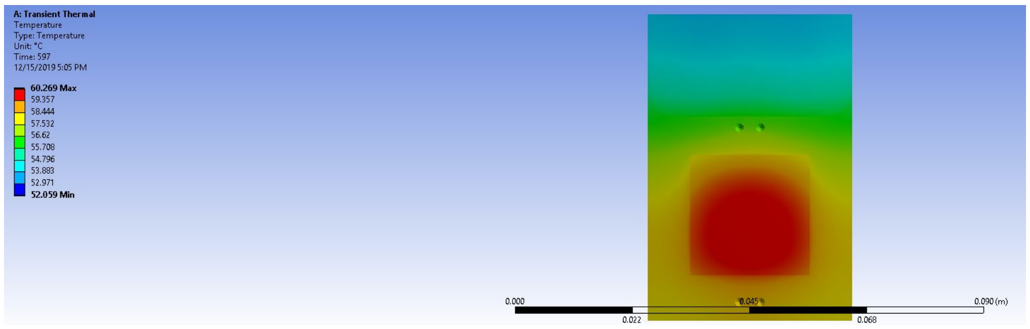
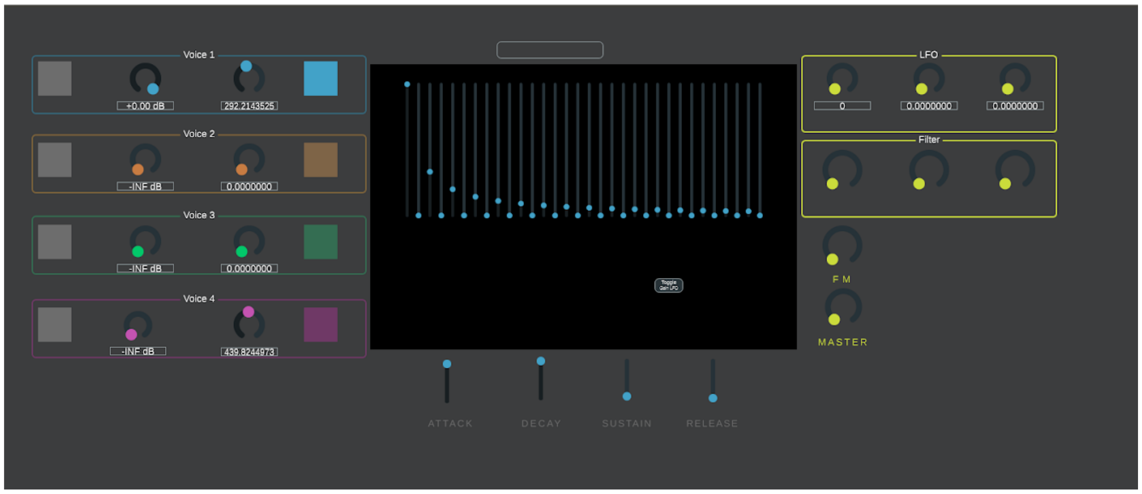


Figure 10: Temperature mapping on bottom of heatsink geometry

The peak temperature reached by the Nano and the heatsink during testing was around 63 degrees Celsius. This is well below the throttling temperature threshold for the essential components. As such, the stock heatsink was determined to have satisfactory performance for thermal constraints, but in order to reach the desired form factor, we needed to use a much smaller design.

The completed GUI design, as shown below, was effective in instantiating state change in the engine and updating its own state in accordance with user input. We were able to achieve control of a unified ADSR envelope across all four voices, as well as real time editing of harmonics for each voice. The base frequency and gain of each voice were adjustable individually. ADSR was implemented for each voice separately, but only worked for one voice. Switching between voices reflected appropriately in the GUI, both in updating the display and changing the ADSR envelope controls to the setting's state. ADSR state was handled in each voice by the engine, making it easier to edit and recall. The Display file tracked the current voice in a pointer which was updated when the 'Select' Button was pressed.



Figure(): Final GUI design created with JUCE

Selecting a different voice brought up the harmonic display for whichever was currently selected. The display interface successfully allowed for the editing of individual harmonic levels. In order to display the level of each voice in decibels, a custom class was created as a subclass of the included 'Slider' component to allow the gain to display in decibels but adjust the linear gain state. The other buttons and sliders implemented the standard component classes included in JUCE, with handlers that called setter and getter functions in the engine to update the state. The select buttons were associated with a radio group, a built in function, to ensure that only one was active at a time.

The GUI was implemented successfully overall, with most bugs appearing to occur in the engine from what we could tell. We failed to implement MIDI input, for which JUCE provides classes and functions. The biggest issue we ran into is that the Linux kernel specially compiled for the Jetson Nano did not seem to include a MIDI driver. With limited time and the difficulty of debugging this feature, we decided to focus instead on furthering the functionality of what we already had. As such, the spacebar key was used to trigger the ADSR and resulting audio output.

Conclusion

We learned a lot through the course of this project. A major conclusion is that using a GPU for audio is more complicated and cumbersome than programming a microcontroller and certain types of DSP chips. While the GPU could have benefits in physically based audio simulation, the specific effects we chose to implement are not the killer apps for GPU audio.

X. Self-learning

Davis

For the past semester, I have self-learned both Digital Signal Processing as well as GPU Audio. I took Digital Signal Processing (ESE 531) which has helped me better understand where our GPU synthesizer failed and what changed will need to be made. All this self learning has shown me that it may be a more useful task to build synthesizers the traditional way and perhaps the GPUs purpose sits within Machine Learning for Signal Process and not within audio generation. I did learn how to work with real-time audio processing on the GPU and I believe that now I can definitively say that any remote advantage that the GPU proposes would be distinctly improved upon in a DSP chip.

Nikil

Looking for components and reading through datasheets has been a challenge. I've been calling some of the manufacturers directly to walk through the datasheets to understand if they support the desired features. This is one of those tasks that just takes a lot of practice to get more proficient and confident, and I definitely learned a good deal both semesters. I've gotten much more familiar with Altium and its hierarchical design features through this project and through my project in ESE516 (IoT electronics). I've been using the compiler and design rules check to spot errors faster than I had done in the past. I also learned a good deal about CUDA and refreshed my signal processing understanding when trying to fix the bugs in the Phase Vocoder that Davis had started developing.

Mason

I spent the first semester learning ANSYS thermal simulation, which I have not explored in previous classes. While my background from Heat and Mass Transfer (MEAM 333) was helpful in knowing what boundary conditions, the structuring and refinement of an accurate simulation was not covered in the course material. I learned how to appropriately navigate ANSYS software, as well as structure a simulation generally.

The second semester provided many more learning opportunities for me. When we pivoted to a pure software project, I first had to learn how to use CMAKE to compile our application. The installation and use of JUCE on a device like the Nano is not exactly common or encouraged, so I also had to learn a lot about the installation of software on a Linux system. JUCE is a set of C++ libraries, a language in which I had zero prior experience. My work in CIS 240 prepared me somewhat for this experience, but I had to learn C++ syntax and the bulk of the GUI libraries provided with JUCE to design and implement our user interface.

Enoch

I spent the first semester self-learning how to program with CUDA, which is an API that allows me to program with the GPU. In addition to this, I learned about music synthesis and luckily for me, there has been quite a bit of overlap with signal processing, a course I've taken here at Penn.

In this last semester, I continued to learn about music synthesis, specifically learning about envelopes (ADSR) and low-frequency oscillators (LFO's). I also learned about socket programming as it relates to communication between different programs on the same machine. This proved to be the fastest form of communication between our software GUI and our synth engine. I also learned how to use the Qt framework, which I had partially known had to use because of a previous course. This project, however, required much more than just the basics I learned in a previous course.

Discussion of Specific Courses

CIS 565 and ESE 350 have been useful in understanding the requirements needed to interact with the GPU and interact with the external world using an embedded system. ESE516 was helpful with PCB design. MEAM 333 has proven useful in giving theoretical foundations for building a thermal simulation. MEAM 545 and MEAM 302 provided background knowledge of simulation in general, and prevented running into common pitfalls and essential steps to ensure the simulation runs well, including mesh considerations and patience. CIS 560, CIS 563, and CIS 380 have been helpful in developing software that is adaptive to different needs and can function in a low level context. ESE 224 and ESE 531 have also been helpful in understanding the music synthesis required for this project.

XI. Ethical and Professional Responsibilities

The societal context of our project is that it will offer producers a cheaper option to current expensive hardware. Regarding the environment, our product will be run off of standard power, either from the wall or from a standard, portable USB C phone battery. These options hold little control over the environmental impact of our device, as they are contingent on the 'green-ness' of the specific power grid.

When considering the production of a music creation device, ethical issues are hard to come by. The product itself is intended to provide an affordable alternative to equipment on the market. The interesting question to consider is the ethics of basing aspects of our design off of existing synthesizer products. While the algorithms we use are originally developed, the interface, general idea, and featureset are inspired by a variety of products including the Teenage Engineering OP1, Ableton Operator, the Nintendo Switch, and a multitude of additive synthesis products. Fortunately, the world of music production equipment tends to be collaborative, and our device differentiates itself enough from other devices to avoid copyright or patent issues.

XII. Meetings

In the first semester we met on a bi-weekly basis throughout the semester with our advisor, Tania Khana. Additionally, stakeholder meetings were conducted through I-Corps, twice a week for five weeks, in order to better understand our market. We met with Tomas Isakowitz about how to conduct product interviews as well as how to better adjust our business model as we learned from the interviews. In the second semester, we continued to meet with Tania on a bi-weekly basis until Spring Break. After spring break, we met with Tania twice.

XIII. Reflection on Fall Milestones and Proposed Spring Schedule

Our major milestones for the fall semester were the implementation of our two features, Additive Synthesis and Phase Vocoder, on the Nano, with control from a physical prototype faceplate. We implemented the phase vocoding algorithm offline on the jetson nano, but were unable to port the additive synthesis engine onto the jetson nano. We also failed to implement active control of the nano, but we have functional I/O processing via JSON using the ESP32 chip for our physical controls.

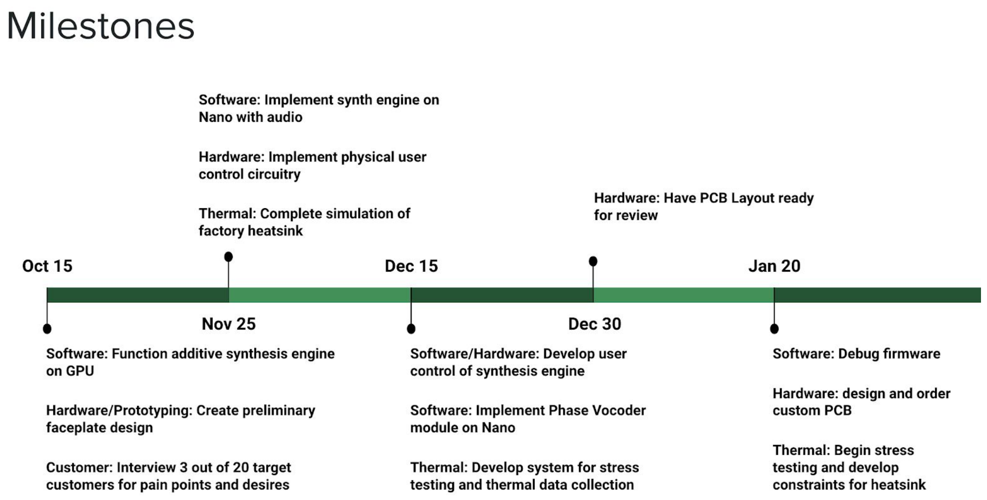

Milestones
Software: Implement synth engine on Nano with audio
Hardware: Implement physical user control circuitry
Thermal: Complete simulation of factory heatsink
Hardware: Have PCB Layout ready for review
Oct 15
Dec 15
Jan 20
Nov 25
Dec 30
Software: Function additive synthesis engine on GPU
Hardware/Prototyping: Create preliminary faceplate design
Customer: Interview 3 out of 20 target customers for pain points and desires
Software/Hardware: Develop user control of synthesis engine
Software: Implement Phase Vocoder module on Nano
Thermal: Develop system for stress testing and thermal data collection
Software: Debug firmware
Hardware: design and order custom PCB
Thermal: Begin stress testing and develop constraints for heatsink

Figure 11: Intended fall and winter milestones

We were able to accomplish some goals we intended in the spring, but ultimately never succeeded in implementing any hardware. Due to indecision on the part of team members responsible for obtaining PCB and other hardware components, we did not have any physical objects to implement prior to COVID-19. After COVID-19, we were able to obtain workstations to continue development of the project. The workstations arrived the Wednesday before the final demo, and as mentioned, required significant setup before any actual work could be done. The entire week leading up to the demo was spent porting the project into JUCE and implementing the new features, including the GUI.

XIV. Discussion of Teamwork

Davis and Enoch executed most of the software implementation. The base of the additive synthesis engine was coded by Davis, and Enoch continued to improve it and add functionality. Nikil handled the electronic hardware, including the prototype control circuit, input data handling, and PCB layout. Mason handled thermal modelling and simulation, as well as laser cutting for the look and feel prototype and faceplate prototype. After COVID-19, Davis and Mason worked on porting the project to JUCE, Enoch continued to add functionality to the Additive Synthesizer Engine, and Nikil continued to work on the Phase Vocoder.

A team conflict we worked to resolve was in design and development of user interaction/experience. Both Nikil and Davis had dramatically different ideas for the look of the faceplate and how users should be able to interact with the additive synthesizer. Davis wanted the layout and utilization of the knobs to be very interactive and have them sectioned off based on their functionality. Nikil wanted the knobs to surround a long screen that would represent the waveforms harmonic content. Based on limitations in screen length as well as the information received from product interviews about how musicians interact with synthesizers we determined that we should modify Davis and Nikil's initial idea into a hybrid design that focuses on users having visual feedback for the function they are currently modifying.

The nature of our project lent itself well to being an inter-departmental team, with clearly delineated tasks that played to our individual strengths. Davis' embedded systems knowledge and general understanding of music hardware proved invaluable, as well as his experience specifically with GPU architecture and programming. Enoch's computer science background and in depth knowledge of the mathematical topics necessary to implement the additive synthesizer allowed him to make significant contributions to that feature. Nikil's electrical engineering and embedded systems experience made his position as hardware lead a natural choice, especially given his familiarity with high speed systems and PCB design. Mason's mechanical background allowed him to understand the thermal problem and construct an accurate simulation in addition to analyzing the structural and thermal concerns of a powerful electronic device in a small form factor.

XV. Budget and Justification

Thus far, we have bought equipment for 4 sets of synthesizers so that each of us can work on our part of the project independently. Our budget has been as follows:

Item	Cost/Unit	# Units	Total Cost
TOTAL			**$1,800**
NVIDIA Jetson Nano	$120	4	$480
5V 4A power supply	$10	4	$40
WiFi Card	$25	4	$100
SD Card	$30	6	$180
Electrical components for modular faceplate prototypes	$70	4	$280
Printed Circuit Board manufacturing costs	$70	4	$280
Case Machining costs	$40	4	$160
ADC/DAC Prototype	$70	4	$280

To produce a full engineering prototype PCB of the controller faceplate, we are looking at PCBWay with parts placed, or 4PCB (which was used by ESE516), or Macrofab (which we would consider using for a full production run). Green Circuits and Macrofab both have the ability to flash the chip on the assembly line and run some basic validation tests before we even get the protoboard back. They are also local in the US. They may also be able to assemble the faceplate casing (with the knobs and aluminum covering).

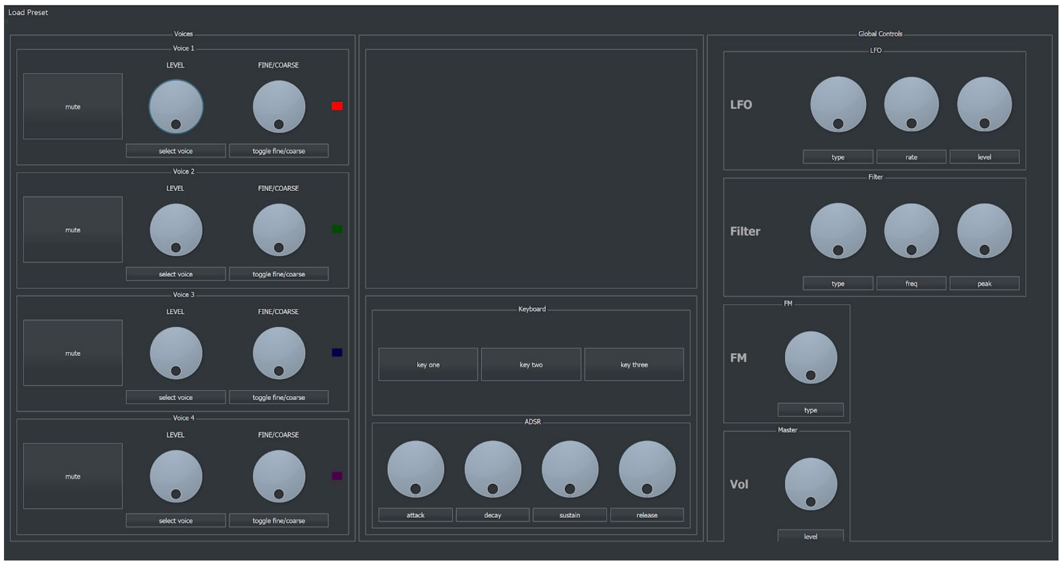
Item	Cost/Unit	# Units	Total Cost
TOTAL			**$5,168**
Electronics Components	$127	4	$508
Mechanical housing cost	$40	4	$160
Tooling cost	$1,000	1	$1,000
Printed Circuit Board Fab cost	$350	6	$2,100
Assembly cost	$350	4	$1,400

XVI. Standards and Compliance

In planning the software, firmware, and hardware for the device (and later, the JUCE implementation), we were constantly making sure we were moving forward to be compatible with current devices and standards. For example, we designed our software to be modular so that it could accept input from the faceplate via JSON packets or via the QT emulator. We additionally wrote the API hooks so that we could take in MIDI input from our faceplate or from an external keyboard. We were following the but packing format for MIDI when developing the Phase Vocoder engine as well, so that we could eventually connect to external devices and play notes for real-time autotune. On the hardware and firmware, we implemented serial communication and selected devices that used I2C.

XVII. Work Done Since Last Semester

In addition to the JUCE GUI, a GUI based on Qt was developed to emulate what would be our physical controls. This GUI was a lot more modular than the JUCE GUI, as the only means of communication was through a JSON packet. An interface had be implemented to allow the synth engine to interpret the commands given through the Qt GUI. In theory, our physical controls would just need to output the same type of JSON packet and the synth engine would work with said controls right away. We had not intended to finish this GUI before the lockdown, as we were planning to move onto the physical controls. But due to losing all of our hardware, we decided on completing the Qt GUI in case the JUCE GUI did not go as intended. This resulted in the following polished Qt GUI.



Figure(): Final Qt GUI

In addition to this Qt GUI, more effects were added to the additive synthesizer to producer richer sounds. The effects in particular were LFO's and ADSR.

XVIII. Discussion and Conclusion

As the project stands, it is a noise generating piece of garbage. Major work would need to be done in order for it to reach a usable or remotely musical state. There are many issues with our current design: aliasing, clipping, and block drop out.

Aliasing occurs whenever we try to produce sinusoids above 22kHz (the Nyquist Frequency of audio sampled at 44.1kHz). The effect of aliasing can be heard when adding, for example, the 64th harmonic of a voice. A ringing sound not present originally is introduced as a product of the sampling rate folding the original waveform down to within the audible range.

Clipping occurs whenever the audio gain surpasses the floating point value of 1. This occurs because all individual sine waves are able to go to gains of 1. This implementation was an oversight that we did not consider until we were able to get multiple waveforms playing simultaneously. Fixing it would entail normalizing the sine waves at each block which is a feasible task that we hope will be accomplished in the future.

Block drop out occurs randomly and without warning resulting in drastic noise and other sound artefacts. We don’t know exactly why this happens, but we assume it’s related to GPU processing.

Hopefully in the future we will be able to design and manufacture a PCB and integrate it with the existing engine. Ideally, this project will someday be a product in the way we had intended back in September.

XIX. Appendices

Our code is rather long (a single file reproduced here would have been 40 pages). You can visit our GitHub repositories at the following links:
https://kutt.it/GPUsynthesizer
https://kutt.it/AutotuneEngine
https://kutt.it/FaceplateSimulator

Selected code is reproduced below and on subsequent pages.

Appendix A: Additive Synthesizer Code

Kernel.h

```
#pragma once


#include <cstdio>
#include <cstring>
#include <cmath>
#include <algorithm>
#include <chrono>
#include <stdexcept>




namespace Additive {
      void initSynth(int numSinusoids, int numSamples, float*
host_frequencies);
      void endSynth();
    void alloc_engine(float2* &h_freq_gains, float* &h_angles, float*
&h_v_gains,
    float* &h_tmp_buffer, float* &h_buffer, float* &h_adsr, bool* &h_v_ons,
int num_samples, int num_voices, int num_harms);
    void compute_sinusoid_gpu_simple(float * buffer, float angle);
    void realloc_engine(float* &h_tmp_buffer,float* &h_buffer, int
prev_num_samples, int num_samples);
    void my_v_compute(float *buffer, float angle, float* h_buffer, float*
h_v_gains, float2* h_freq_gains, float* h_adsr, bool* h_v_ons, int numSamples,
int numSinusoids, int numVoices);
    void compute_sinusoid_hybrid(float* samples, float2* h_freq_gains, float*
h_angles, float *h_v_gains, float* h_tmp_buffer, float* h_buffer, int
numSinusoids, float time, float numSamples);
}
```

Kernel.cu

```
#include "kernel.h"
#include <math.h>
#include <cmath>
#include <stdio.h>
#include <cuda.h>
#include <iostream>
//divide
```

```
#define THREADS_PER_SAMPLE 16
#define SAMPLES_PER_THREAD 1
#define SAMPLING_FREQ 44100
//#define SIMPLE 0
#define checkCUDAErrorWithLine(msg) checkCUDAError(msg, __LINE__)
float *dev_buffer, *dev_tmp_buffer;
float slideTime;
int numSamples, numSinusoids, numVoices;
void printArraywNewLines(int n, float *a, bool abridged) {
    printf("    [ ");
    for (int i = 0; i < n; i++) {
        if (abridged && i + 2 == 15 && n > 16) {
            i = n - 2;
            printf("... ");
        }
        printf("%3f\n", a[i]);
    }
    printf("]\n");
}
void printArraywNewLines(int n, float2 *a, bool abridged) {
    printf("    [ ");
    for (int i = 0; i < n; i++) {
        if (abridged && i + 2 == 15 && n > 16) {
            i = n - 2;
            printf("... ");
        }
        printf("%3f, ", a[i].x);
        printf("%3f\n", a[i].y);
    }
    printf("]\n");
}
void printArray(int n, float2 *a, bool abridged) {
    printf("    [ ");
    for (int i = 0; i < n; i++) {
        if (abridged && i + 2 == 15 && n > 16) {
            i = n - 2;
            printf("... ");
        }
        printf("{%3f, ", a[i].x);
        printf("%3f},", a[i].y);
    }
    printf("]\n");
}

/**
* Check for CUDA errors; print and exit if there was a problem.
*/
void checkCUDAError(const char *msg, int line = -1) {
  cudaError_t err = cudaGetLastError();
  if (cudaSuccess != err) {
    if (line >= 0) {
      fprintf(stderr, "Line %d: ", line);
    }
    fprintf(stderr, "Cuda error: %s: %s.\n", msg, cudaGetErrorString(err));
    exit(EXIT_FAILURE);
  }
}

void Additive::alloc_engine(float2* &h_freq_gains, float* &h_angles, float*
&h_v_gains, float* &h_tmp_buffer,
 float* &h_buffer,float* &h_adsr, bool* &h_v_ons, int num_samples, int
num_voices, int num_harms){
      cudaHostAlloc((void**)&h_freq_gains, sizeof(float2)*num_harms *
num_voices, cudaHostAllocMapped);
     checkCUDAError("h_freq_gains get Device Pointer", __LINE__);

     cudaHostAlloc((void**)&h_angles, sizeof(float)*num_harms*num_voices,
cudaHostAllocMapped);
     checkCUDAError("h_angles get Device Pointer", __LINE__);

     cudaHostAlloc((void**)&h_v_gains, sizeof(float)*num_voices,
cudaHostAllocMapped);
     checkCUDAError("h_v_gains get Device Pointer", __LINE__);

     cudaHostAlloc((void**)&h_tmp_buffer, sizeof(float)*num_samples,
cudaHostAllocMapped);
     checkCUDAError("h_tmp_buffer get Device Pointer", __LINE__);

     cudaHostAlloc((void**)&h_buffer, sizeof(float)*num_samples,
cudaHostAllocMapped);
     checkCUDAError("h_buffer get Device Pointer", __LINE__);

     cudaHostAlloc((void**)&h_adsr, sizeof(float)*num_voices*num_samples,
cudaHostAllocMapped);
     checkCUDAError("h_Adsrs get Device Pointer", __LINE__);

     cudaHostAlloc((void**)&h_v_ons, sizeof(bool)*num_voices,
cudaHostAllocMapped);
     checkCUDAError("h_v_ons get Device Pointer", __LINE__);
}

void Additive::realloc_engine(float* &h_tmp_buffer,float* &h_buffer, int
prev_num_samples, int num_samples){
          float tmp_buffer[num_samples];
              float buffer[num_samples];
          cudaMemcpy(tmp_buffer, h_tmp_buffer, sizeof(float)*prev_num_samples,
cudaMemcpyHostToHost);
              checkCUDAError("memcpy realloc firs tmp_buffer", __LINE__);
              cudaMemcpy(buffer, h_buffer, sizeof(float)*prev_num_samples,
cudaMemcpyHostToHost);
              checkCUDAError("memcpy realloc first buffer", __LINE__);
              cudaFree(h_tmp_buffer);
              cudaFree(h_buffer);
          cudaHostAlloc((void**)&h_tmp_buffer, sizeof(float)*num_samples,
cudaHostAllocMapped);
          checkCUDAError("h_tmp_buffer get Device Pointer", __LINE__);
          cudaHostAlloc((void**)&h_buffer, sizeof(float)*num_samples,
cudaHostAllocMapped);
```

```
          checkCUDAError("h_buffer get Device Pointer", __LINE__);

              cudaMemcpy(h_tmp_buffer,tmp_buffer,
sizeof(float)*prev_num_samples, cudaMemcpyHostToHost);
                  checkCUDAError("memcpy realloc tmp_buffer", __LINE__);
              cudaMemcpy(h_buffer,buffer,  sizeof(float)*prev_num_samples,
cudaMemcpyHostToHost);
                  checkCUDAError("memcpy reallox buffer", __LINE__);

}
__global__ void my_vh_kernel(float *outBuffer, float2 *freq_gains, float
*vgains, float* adsr, bool* vons, float angle, int numSamples, int
numSinusoids, int numVoices)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;

      if (idx < numSamples) {
            // samples sine wave in discrete steps
            angle = angle + 2.f * M_PI * idx / 44100.f;

            float buff_val = 0.f;
            int numHarmonics = numSinusoids / numVoices;

            for (int i = 0; i < numVoices; i++) {
                  for (int j = 0; j < numHarmonics; j++) {
                        buff_val += vons[i] * vgains[i] * adsr[i * idx] *
freq_gains[i*numHarmonics + j].y * __sinf(angle * freq_gains[i*numHarmonics +
j].x);
                        //printf("idx %d buff val: %f\n", idx, buff_val);
                  }
            }

            // buffer to be sent to DAC
            outBuffer[idx] = buff_val;
      }
}

void Additive::my_v_compute(float *buffer, float angle, float* h_buffer,
float* h_v_gains,
     float2* h_freq_gains, float* h_adsr, bool* h_v_ons, int numSamples,
       int numSinusoids, int numVoices)
      {
            //static int count = 0;
            //std::cout << "frequency" << std::endl;
            //printArray(1, h_freq_gains, 0);
            int threadsPerBlock = numSamples;
            int blocksPerGrid = (numSamples + threadsPerBlock - 1) /
threadsPerBlock;
            float *dev_buffer, *dev_v_gains, *dev_adsr;
          bool *dev_v_ons;
            float2* dev_freq_gains;
            cudaHostGetDevicePointer((void**)&dev_freq_gains,
(void*)h_freq_gains, 0);
            checkCUDAError("dev_freq_gains get Device Pointer", __LINE__);
            //cudaHostGetDevicePointer((void**)&dev_angles, (void*)h_angles,
0);
            cudaHostGetDevicePointer((void**)&dev_v_gains, (void*)h_v_gains,
0);
            checkCUDAError("dev_v_gains get Device Pointer", __LINE__);

            cudaHostGetDevicePointer((void**)&dev_buffer, (void*)h_buffer, 0);
            checkCUDAError("dev_buffer get Device Pointer", __LINE__);

            cudaHostGetDevicePointer((void**)&dev_adsr, (void*)h_adsr, 0);
            checkCUDAError("dev_adsr get Device Pointer", __LINE__);

          cudaHostGetDevicePointer((void**)&dev_v_ons, (void*)h_v_ons, 0);
            checkCUDAError("dev_v_ons get Device Pointer", __LINE__);


            my_vh_kernel <<< blocksPerGrid, threadsPerBlock >>> (dev_buffer,
dev_freq_gains, dev_v_gains, dev_adsr, dev_v_ons,
                                                              angle, numSamples,
numSinusoids, numVoices);
            checkCUDAError("vhkernel error", __LINE__);
            cudaStreamSynchronize(NULL);
                        //std::cout << "reyeet"<<std::endl;
            #ifdef KERNELDEBUG
           float *debug_arr1;
          cudaMallocManaged((void**)&debug_arr1, sizeof(float) * numSamples,
cudaMemAttachHost);
          checkCUDAError("Error debugging output after cufftshift (malloc)",
__LINE__);
          cudaMemcpy(debug_arr1,dev_buffer, sizeof(float)
*numSamples,cudaMemcpyDeviceToHost);
          checkCUDAError("Error debugging output after cufftshift (memcpy)",
__LINE__);
          printf("out\n");
          printArraywNewLines(numSamples, debug_arr1, 0);
          cudaFree(debug_arr1);
            #endif
    //std::cout << "yeet" << std::endl;
            // updates the buffer with dev_buffer computed in GPU
            cudaMemcpy(buffer, dev_buffer, numSamples * sizeof(float),
cudaMemcpyDeviceToHost);
            checkCUDAError("memcpy error", __LINE__);
            #ifdef DEBUGCOPYM
            std::cout << "kernel" << std::endl;
              float *debug_arr1;
          cudaMallocManaged((void**)&debug_arr1, sizeof(float) * numSamples,
cudaMemAttachHost);
          checkCUDAError("Error debugging output after cufftshift (malloc)",
__LINE__);
         memcpy(debug_arr1,buffer, sizeof(float) *numSamples);
          checkCUDAError("Error debugging output after cufftshift (memcpy)",
__LINE__);
          printf("out\n");
          printArraywNewLines(numSamples, debug_arr1, 1);
          cudaFree(debug_arr1);
            #endif


}


__device__ float ramp_kern(float currentTime, float slideTime, float f0, float 
f1){
      float integral;
      if (currentTime < slideTime) {
            float k = (f1-f0) / slideTime;
            integral = currentTime * (f0 + k * currentTime / 2.0f);
      } else {
            integral = f0 * slideTime + (f1 - f0) * slideTime / 2.0f;
            integral += (currentTime - slideTime) * f1;
      }
      return integral * 2.0f * M_PI;
}

#define imin( a, b ) ( ((a) < (b)) ? (a) : (b) )

__global__ void sin_kernel_fast(float2* freq_gains, float* buffer,
                                                float* angles, int 
numThreadsPerBlock, int numSinusoids,
                                                float time, int numSamples)
{
      int idx = blockIdx.x * blockDim.x + threadIdx.x;

      if (idx < numSamples * THREADS_PER_SAMPLE) {
            //determine how many sineWaves are to be computed in each thread 
based on how many threads it takes to compute a sample
            int maxSinePerBlock = (numSinusoids + THREADS_PER_SAMPLE - 1) / 
THREADS_PER_SAMPLE;
            int sinBlock = idx / numThreadsPerBlock;
            int sampleIdx = idx - sinBlock * numThreadsPerBlock; // modulo 
function but GPUs are trash at modulo so don't use it
            float val[SAMPLES_PER_THREAD];
            for (int j = 0; j < SAMPLES_PER_THREAD; j++) {
                  val[j] = 0.0f;
            }
          float gain, freq0, angle, angleStart;
          int firstSine = sinBlock * maxSinePerBlock;
            int lastSine = imin(numSinusoids, firstSine + maxSinePerBlock);
            //compute samples for maxSinePerBlock
            for (int i = firstSine; i < lastSine; i++) {
                  angleStart = angles[i];
                  freq0 = freq_gains[i].x;
                  gain = freq_gains[i].y;
                  for (int j = 0; j < SAMPLES_PER_THREAD; j++) {
                    angle = angleStart + time + 
(sampleIdx*SAMPLES_PER_THREAD+j) / SAMPLING_FREQ;
                        val[j] += __sinf(angle * freq0) * gain / numSinusoids;
                  }
                  angles[i] = fmod(angle, 44100.f);
            }
            for (int i = 0; i < SAMPLES_PER_THREAD; i++) {
                  buffer[idx * SAMPLES_PER_THREAD + i] = val[i];
            }

      }

}


__global__ void sum_blocks(float* tmp_buffer, float* buffer, int numSamples) {
      int idx = blockIdx.x * blockDim.x + threadIdx.x;

      if (idx < numSamples) {
            float sum = 0;
            for (int i = 0; i < THREADS_PER_SAMPLE; i++) {
                  sum += tmp_buffer[idx + i * numSamples];
            }
            buffer[idx] = sum;
      }

}



void Additive::compute_sinusoid_hybrid(float* samples, float2* h_freq_gains,
float* h_angles, float *h_v_gains, float* h_tmp_buffer, float* h_buffer, int
numSinusoids, float time, float numSamples){
      int threadsPerBlock = 256;
      int numThreadsPerBlock = numSamples / SAMPLES_PER_THREAD;
      int numThreads = THREADS_PER_SAMPLE * numThreadsPerBlock;
      int blocksPerGrid = (numThreads + threadsPerBlock - 1) /
threadsPerBlock;
      float2* dev_freqs_gains;
      float* dev_buffer,* dev_tmp_buffer, *dev_angles, *dev_v_gains;
      cudaHostGetDevicePointer((void**)&dev_freqs_gains, (void*)h_freq_gains,
0);
      checkCUDAError("dev_freq_gains get Device Pointer", __LINE__);
      cudaHostGetDevicePointer((void**)&dev_angles, (void*)h_angles, 0);
      checkCUDAError("dev_angles get Device Pointer", __LINE__);
      cudaHostGetDevicePointer((void**)&dev_v_gains, (void*)h_v_gains, 0);
      checkCUDAError("dev_v_gains get Device Pointer", __LINE__);
      cudaHostGetDevicePointer((void**)&dev_tmp_buffer, (void*)h_tmp_buffer,
0);
      checkCUDAError("dev_tmp_buffer get Device Pointer", __LINE__);
      cudaHostGetDevicePointer((void**)&dev_buffer, (void*)h_buffer, 0);
      checkCUDAError("dev_buffer get Device Pointer", __LINE__);


      sin_kernel_fast <<<blocksPerGrid, threadsPerBlock >>>(dev_freqs_gains,
dev_tmp_buffer, dev_angles, numThreadsPerBlock, numSinusoids, time,
numSamples);
```

```
      //checkCUDAErrorWithLine("sin_kernel_fast failed");
      blocksPerGrid = (numSamples + threadsPerBlock - 1) / threadsPerBlock;
      sum_blocks <<<blocksPerGrid, threadsPerBlock >> >(dev_tmp_buffer,
dev_buffer, numSamples);
      //checkCUDAErrorWithLine("sum_blocks failed");
      cudaMemcpy(samples, dev_buffer, numSamples * sizeof(float),
cudaMemcpyDeviceToHost);
}
```

Engine.h

```
#pragma once

#include "constants.h"
#include "kernel/kernel.h"
#include <iostream>
#include <JuceHeader.h>
#include "SynthADSR.h"
#include "lfo.h"

class Engine {
    private:
        static Engine *engine;

        float2 *h_freq_gains;    // contains freqs for every harmonic of every
voice
        float *h_angles;
        float *h_v_gains;      // gain for overall voice
        float *h_buffer;
        float *h_tmp_buffer;
        float *h_adsr;
        float *samples;
        float *fundamental_freqs;
        float time;
        bool *h_v_ons;       // toggle on-off (mute status) without changing
gain
        SynthADSR *adsr[4];
        LFO *gain_lfo;
        int enable_gain_lfo;

        int *freq_ratios;
        int num_samples;
        int num_sinusoids;
        int num_harms;
        int num_voices;
        float angle = 0;

        Engine(int num_samples);
        ~Engine();
        Engine(Engine const&){};
        Engine& operator=(Engine const&){};
        void update_freqs();
        void realloc_engine(int num_samples);
    public:
        // returns the singleton instance
        static Engine* getInstance();
        static Engine* getInstance(int num_samples);

        // output generator
        void tick(void* outputBuffer);
        void simple_tick(void* outputBuffer, int num_Samples);

        void load_sinewave(int v_idx, int f);
        void load_sawtooth(int v_idx, int f);
        void load_square_wave(int v_idx, int f);

        void update_fundamental(int v_idx, float freq);
        void update_voice_gain(int v_idx, float gain);
        void update_harmonics(int v_idx, int harmonic, float gain);
        void toggleMute(int v_idx);
        float get_freq(int v_idx, int harmonic);
        float get_gain(int v_idx, int harmonic);
        bool get_mute(int v_idx);


        // ADSR functionality
        void gate_on();
        void gate_off();
        void process_adsr(void *outputBuffer);
        void get_adsr(int v_idx, float* curr_adsr);
        void set_adsr(int v_idx, float* curr_adsr);


        // LFO functionality
        void process_gain_lfo(void *outputBuffer, float angle);
        void set_gain_lfo_rate(float rate);
        void set_gain_lfo_level(float level);
        void set_gain_lfo_type(float knob_val, float max_val);

        void toggle_gain_lfo();
};
```

Engine.cpp

```
using namespace std;
#include "engine.h"


Engine* Engine::engine = NULL;


Engine::Engine(int num_samples) {

    gain_lfo = new LFO();
    gain_lfo->set_level(0.f);
    gain_lfo->set_rate(0.f);
    enable_gain_lfo = 1;

    for ( int i = 0; i < 4; i++){
        adsr[i] = new SynthADSR();
        adsr[i]->setAttackRate(.1 * SAMPLING_FREQUENCY);    // .1 seconds
        adsr[i]->setDecayRate(.3 * SAMPLING_FREQUENCY);     // .3 seconds
        adsr[i]->setReleaseRate(5 * SAMPLING_FREQUENCY);    // 5 seconds
        adsr[i]->setSustainLevel(.5);
    }

    // initialize adsr settingsinclude cud
    num_voices= NUM_VOICES_INIT;
    num_harms = NUM_HARMS_INIT;
    num_sinusoids= num_harms * num_voices;
    this->num_samples = num_samples;

    Additive::alloc_engine(h_freq_gains, h_angles, h_v_gains, h_tmp_buffer,
                           h_buffer, h_adsr, h_v_ons, num_samples, num_voices,
num_harms);

    for(int i = 0; i < num_voices; i++){
        h_v_gains[i]  = 1.0;
        h_v_ons[i] = false;
    }

    // turn on 1 voice to start so it makes some kind of sound
    // h_v_ons[1] = true; // we can turn this on once all the harmonics bugs
are fixed

    load_square_wave(0, 440);
    load_sawtooth(1,440);
    load_sinewave(2,440);
}

void Engine::realloc_engine(int num_samples){

     Additive::realloc_engine(h_buffer, h_tmp_buffer,this->num_samples,
num_samples);
}
Engine* Engine::getInstance(int num_samples){
     if(!Engine::engine) Engine::engine = new Engine(num_samples);
     else if (num_samples != Engine::engine->num_samples) {
         engine->realloc_engine(num_samples);
     }
     return Engine::engine;
}

void Engine::toggleMute(int v_idx){
     h_v_ons[v_idx] = !h_v_ons[v_idx];
}

Engine* Engine::getInstance(){
     if(!Engine::engine) Engine::engine = new Engine(128);
     return Engine::engine;
```

```
}

void Engine::process_adsr(void* outputBuffer){
   // adsr->batch_process(NUM_SAMPLES, (float*)outputBuffer);
}

void Engine::load_sawtooth(int v_idx, int f) {

     float L = 1;
     update_fundamental(v_idx, f);
     for (int i = 0; i < num_harms; i++) {
          h_freq_gains[v_idx*num_harms + i].y = (-1.f / (_PI * (i + 1)));
     }
}
void Engine::load_square_wave(int v_idx, int f) {
     update_fundamental(v_idx, f);
     for (int i = 0; i < num_harms; i+=2) {
          h_freq_gains[v_idx*num_harms + i].y = 1.f / (1.f + (i)); //gain

     }
}


void Engine::load_sinewave(int v_idx, int f) {
          h_freq_gains[v_idx*num_harms].y = 1.0; //gain
          h_freq_gains[v_idx*num_harms].x = f;   //freq

}
void Engine::update_freqs(){
     for (int i = 0; i < num_voices; i++){
        for (int j  = 0; j < num_harms; j++){
             h_freq_gains[i*num_harms + j].x = freq_ratios[i*num_harms
+j]*fundamental_freqs[i];
        }
     }

}
void Engine::update_voice_gain(int v_idx, float gain){
     h_v_gains[v_idx] = gain;

}

void Engine::update_fundamental(int v_idx, float freq){
     h_freq_gains[v_idx * num_harms].y = 1.0f;
     for(int i = 0; i < num_harms; i++){
          h_freq_gains[v_idx * num_harms + i].x = freq*(i+1);
     }
}

void Engine::update_harmonics(int v_idx, int harmonic, float gain){
     h_freq_gains[v_idx*num_harms + harmonic].y = gain;
}

float Engine::get_freq(int v_idx, int harmonic){
     return h_freq_gains[v_idx*num_harms + harmonic].x;
}

float Engine::get_gain(int v_idx, int harmonic){
     return h_freq_gains[v_idx*num_harms + harmonic].y;
}

bool Engine::get_mute(int v_idx)
{
     return h_v_ons[v_idx];
}

void Engine::get_adsr(int v_idx, float* curr_adsr)
{
     curr_adsr[0] = adsr[v_idx]->get_atk();
     curr_adsr[1] = adsr[v_idx]->get_dec();
     curr_adsr[2] = adsr[v_idx]->get_stn();
     curr_adsr[3] = adsr[v_idx]->get_rel();

}

void Engine::set_adsr(int v_idx, float* new_adsr)
{
     adsr[v_idx]->setAttackRate(new_adsr[0]);
     adsr[v_idx]->setDecayRate(new_adsr[1]);
     adsr[v_idx]->setSustainLevel(new_adsr[2]);
     adsr[v_idx]->setReleaseRate(new_adsr[3]);
}
void Engine::gate_on(){
     for(int i =0 ; i < 4;i++){
          adsr[i]->gate(ON_G);
     }
}

void Engine::gate_off(){
      for(int i =0 ; i < 4;i++){
          adsr[i]->gate(OFF_G);
     }
}

// ------------ lfo functions ------------- //

void Engine::process_gain_lfo(void *outputBuffer, float angle) {
    if (enable_gain_lfo) {
        gain_lfo->batch_gain_process(NUM_SAMPLES, (float*)outputBuffer,
angle);
    }
}

void Engine::set_gain_lfo_rate(float rate) {
    gain_lfo->set_rate(rate);
}
```

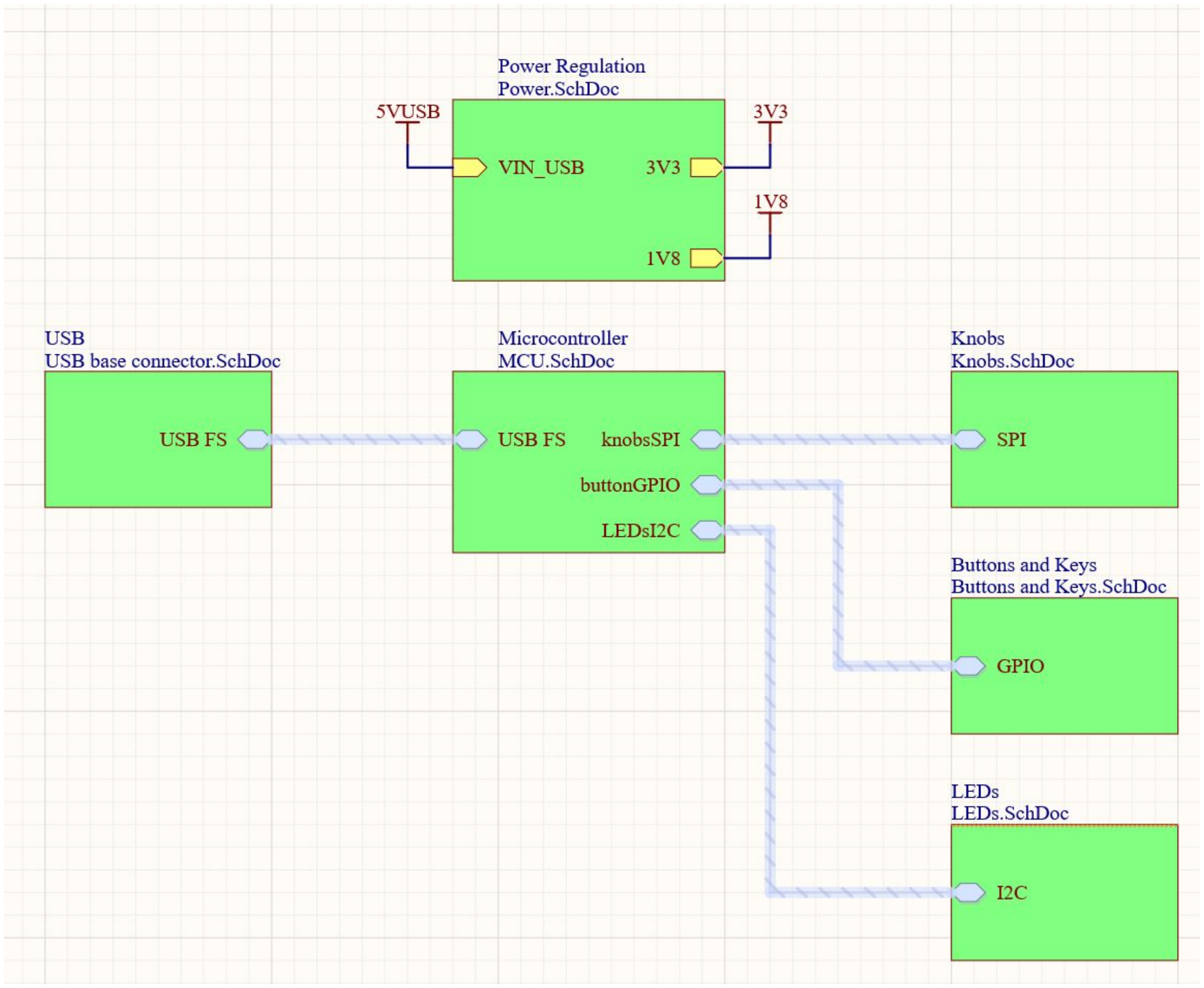
```
void Engine::set_gain_lfo_level(float level) {
    gain_lfo->set_level(level);
}

void Engine::set_gain_lfo_type(float knob_val, float max_val) {
    float val = (knob_val / max_val) * NUM_LFO_WAVES;
    val = std::floor(std::max(0.f, val - 0.01f));
    int type = int(val);
    gain_lfo->set_type(type);
}

void Engine::toggle_gain_lfo() {
    enable_gain_lfo = !enable_gain_lfo;
}



void Engine::tick(void* outputBuffer){
     Additive::compute_sinusoid_hybrid((float*)outputBuffer, h_freq_gains,
h_angles, h_v_gains, h_tmp_buffer, h_buffer,num_sinusoids, time,
this->num_samples);
     time += NUM_SAMPLES / 44100.f;

}

void Engine::simple_tick(void *outputBuffer, int num_Samples){
    // std::cout<< "tick" <<std::endl;
    for(int i = 0; i < 4; i++){
         adsr[i]->process_SynthADSR(this->num_samples, &h_adsr[i *
this->num_samples]);
    }
     Additive::my_v_compute((float*)outputBuffer, angle,
     h_buffer,h_v_gains, h_freq_gains, h_adsr, h_v_ons, this->num_samples,
num_sinusoids, num_voices);

     process_gain_lfo((float*) outputBuffer, angle);
     angle += MathConstants<float>::twoPi * this->num_samples  / 44100.f;
}
```

Appendix B: Phase Vocoder Code

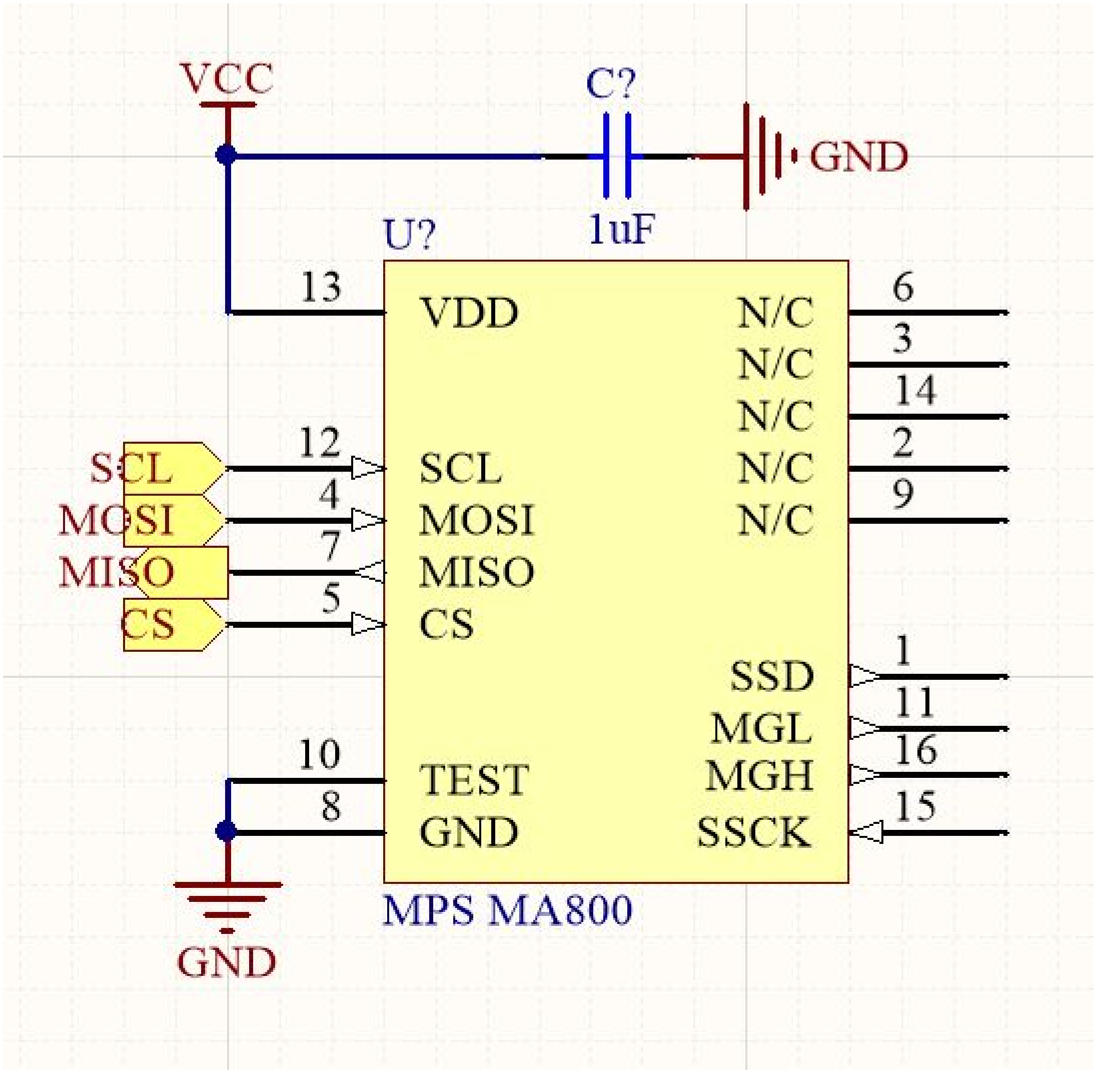
```
 __global__ void  cudaTimeScale(float2* input, int N, int timeScale) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
      if (idx >  N) {
            return;
      }

      input[idx].x = input[idx].x * cosf(timeScale * input[idx].y);
      input[idx].y = input[idx].x * sinf(timeScale * input[idx].y);
   }
__global__ void cufftShift(float2* output, float* input, int N){
    int idx = blockIdx.x *  blockDim.x + threadIdx.x;
    if(idx >= N / 2){
      return;
    }
    output[idx].x = input[idx + N/2];
    output[idx + N/2].x = input[idx];
}
 __global__ void cudaMagFreq(float2* output, float2* input, int N){
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx >= N){
      return;
    }
    output[idx].x = sqrtf(input[idx].x * input[idx].x + input[idx].y *
input[idx].y);
    output[idx].y = atanf(input[idx].y / input[idx].x);
  }


__global__ void cudaWindow(float* input, float* win, int nSamps, int offset){
    int idx = blockIdx.x *  blockDim.x + threadIdx.x;
    if (idx >= nSamps){
        return;
    }
    input[idx + offset] = input[idx + offset] * win[idx];
  }

void pv_analysis_CUFFT(float2* output, float2* fft, float* input, float*
intermediary, float* win, int N) {
      timer().startGPUTimer();
      cudaWindow<< <1,N >> > (input, intermediary, win, N);
      #ifdef DEBUGwindow
          float *debug_arr;
          cudaMallocManaged((void**)&debug_arr, sizeof(float) * N,
cudaMemAttachHost);
          cudaMemcpy(debug_arr,input, sizeof(float) *
N,cudaMemcpyDeviceToHost);
          printf("in\n");
          printArraywNewLines(N, debug_arr);
          cudaMemcpy(debug_arr,intermediary, sizeof(float) *
N,cudaMemcpyDeviceToHost);
          printf("intermediary\n");
          printArraywNewLines(N, debug_arr);
          cudaFree(debug_arr);
      #endif
              checkCUDAError_("Window analysis", __LINE__);
              cufftShiftPadZeros<<<1, N/2>>>(output, intermediary, N, N);
      #ifdef DEBUGpad
          float2 *debug_arr1;
          cudaMallocManaged((void**)&debug_arr1, sizeof(float2) * N,
cudaMemAttachHost);
```

```
          cudaMemcpy(debug_arr1,output, sizeof(float2) *
N,cudaMemcpyDeviceToHost);
          printf("out\n");
          printArraywNewLines(N, debug_arr1);
          cudaFree(debug_arr1);
      #endif
      checkCUDAError_("pad zero analysis", __LINE__);
      cufftHandle plan;
              cufftPlan1d(&plan, 2 * N, CUFFT_C2C, 1);
              cufftExecC2C(plan, (cufftComplex *)output, (cufftComplex
*)output, CUFFT_FORWARD);
              checkCUDAError_("Cufft Error analysis", __LINE__);
      #ifdef DEBUGCUFFT
          float2 *debug_arr2;
          cudaMallocManaged((void**)&debug_arr2, sizeof(float2) *2* N,
cudaMemAttachHost);
          cudaMemcpy(debug_arr2,output, sizeof(float2) *2*
N,cudaMemcpyDeviceToHost);
          printf("postcufft\n");
          printArraywNewLines(2*N, debug_arr2);
          cudaFree(debug_arr2);
      #endif
      cufftDestroy(plan);
      cudaMagFreq << <1,2 * N >> > (output,  2*N);
      checkCUDAError_("magfreq Error analysis", __LINE__);
      #ifdef DEBUGMAG
          float2 *debug_arr3;
          cudaMallocManaged((void**)&debug_arr3, sizeof(float2) *2 * N,
cudaMemAttachHost);
          cudaMemcpy(debug_arr3,output, sizeof(float2) *2 *
N,cudaMemcpyDeviceToHost);
          printf("postMagnitude\n");
          printArraywNewLines(2*N, debug_arr3);
          cudaFree(debug_arr3);
      #endif
      timer().endGPUTimer();
    }
      //#define DEBUGTS
      //#define DEBUGIFFT
      //#define DEBUGSHIFTRE
            void resynthesis_CUFFT(float* output, float* backFrame, float2*
frontFrame, float* win,int N, int hopSize) {
      timer().startGPUTimer();
                  cudaTimeScale << <1,2*N >> > (frontFrame,2* N, 1);
      #ifdef DEBUGTS
          float2 *debug_arr2;
          cudaMallocManaged((void**)&debug_arr2, sizeof(float2) * 2 * N,
cudaMemAttachHost);
          cudaMemcpy(debug_arr2,frontFrame, sizeof(float2) * 2 *
N,cudaMemcpyDeviceToHost);
          printf("postTS\n");
          printArraywNewLines(N, debug_arr2);
          cudaFree(debug_arr2);
      #endif
      cufftHandle plan;
              cufftPlan1d(&plan,  N, CUFFT_C2R, 1);
              checkCUDAError_("Cufft Plan IFFT Error", __LINE__);
                  cufftExecC2R(plan, (cufftComplex*)frontFrame, (cufftReal
*)output);
                  checkCUDAError_("ifft error");
      cufftDestroy(plan);
                  checkCUDAError_("cufftDestory error");
      #ifdef DEBUGIFFT
          float *debug_arr;
          cudaMallocManaged((void**)&debug_arr, sizeof(float) *  N,
cudaMemAttachHost);
          checkCUDAError_("Error debugging output after ifft (malloc)",
__LINE__);
          cudaMemcpy(debug_arr,output, sizeof(float) *
N,cudaMemcpyHostToHost);
          checkCUDAError_("Error debugging output after ifft (memcpy)",
__LINE__);
          printf("CU IFFT\n");
          printArraywNewLines(N, debug_arr);
          cudaFree(debug_arr);
      #endif
                  cudaDivVec << <1, N >> > (output, N, N);
      checkCUDAError_("divvec error");
      #ifdef DEBUGSCALE
          float *debug_arr1;
          cudaMallocManaged((void**)&debug_arr1, sizeof(float) * N,
cudaMemAttachHost);
          checkCUDAError_("Error debugging output after ifft (malloc)",
__LINE__);
          cudaMemcpy(debug_arr1, output, sizeof(float) *
N,cudaMemcpyHostToHost);
          checkCUDAError_("Error debugging output after ifft (memcpy)",
__LINE__);
          printf("SCALE RE\n");
          printArraywNewLines(N, debug_arr1);
          cudaFree(debug_arr1);
      #endif

                  cufftShift<<<1,N/2>>>(output, N);
                  checkCUDAError_("shift error");
      #ifdef DEBUGSHIFTRE
          float *debug_arr3;
          cudaMallocManaged((void**)&debug_arr3, sizeof(float) * N,
cudaMemAttachHost);
          checkCUDAError_("Error debugging output after ifft (malloc)",
__LINE__);
          cudaMemcpy(debug_arr3, output, sizeof(float) *
N,cudaMemcpyHostToHost);
          checkCUDAError_("Error debugging output after ifft (memcpy)",
__LINE__);
          printf("SHIFT RE\n");
          printArraywNewLines(N, debug_arr3);
          cudaFree(debug_arr3);
      #endif

                  cudaWindow<<<1, N>>>(output, win, N);
                  checkCUDAError_("window error");
      #ifdef DEBUGSHIFTRE
          float *debug_arr4;
          cudaMallocManaged((void**)&debug_arr4, sizeof(float) * N,
cudaMemAttachHost);
          checkCUDAError_("Error debugging output after ifft (malloc)",
__LINE__);
          cudaMemcpy(debug_arr4, output, sizeof(float) *
N,cudaMemcpyHostToHost);
          checkCUDAError_("Error debugging output after ifft (memcpy)",
__LINE__);
          printf("WINDOW resynth\n");
          printArraywNewLines(N, debug_arr4);
          cudaFree(debug_arr4);
      #endif

                  cudaOverlapAdd<<<1,N>>>(backFrame, output, N, hopSize);
                  checkCUDAError_("add error");
      #ifdef DEBUGOADD
         float *debug_arr5;
          cudaMallocManaged((void**)&debug_arr5, sizeof(float) * N,
cudaMemAttachHost);
          checkCUDAError_("Error debugging output after ifft (malloc)",
__LINE__);
          cudaMemcpy(debug_arr5, output, sizeof(float) *
N,cudaMemcpyHostToHost);
          checkCUDAError_("Error debugging output after ifft (memcpy)",
__LINE__);
          printf("WINDOW resynth\n");
          printArraywNewLines(N, debug_arr5);
          cudaFree(debug_arr5);
      #endif
      timer().endGPUTimer();
      }
```

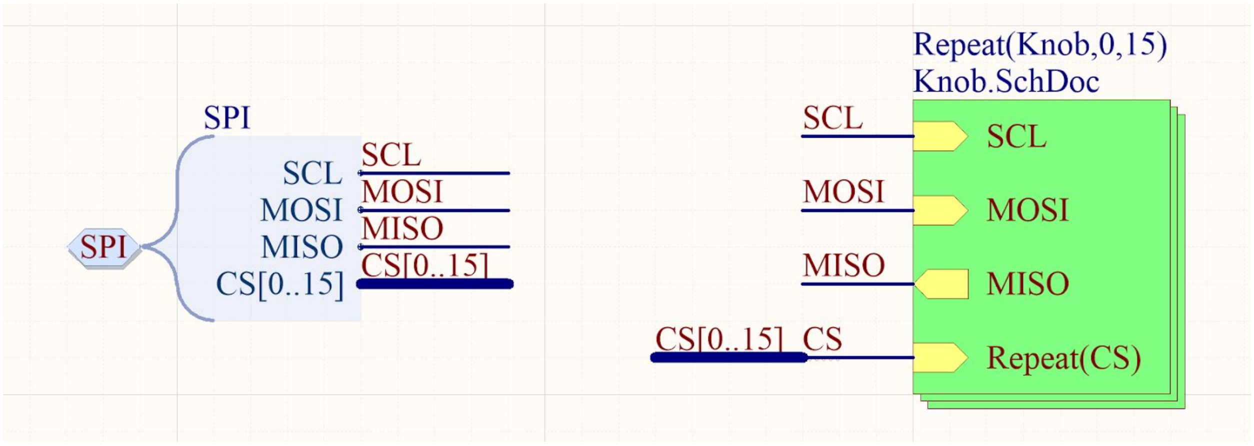
Appendix D: Faceplate PCB Diagram



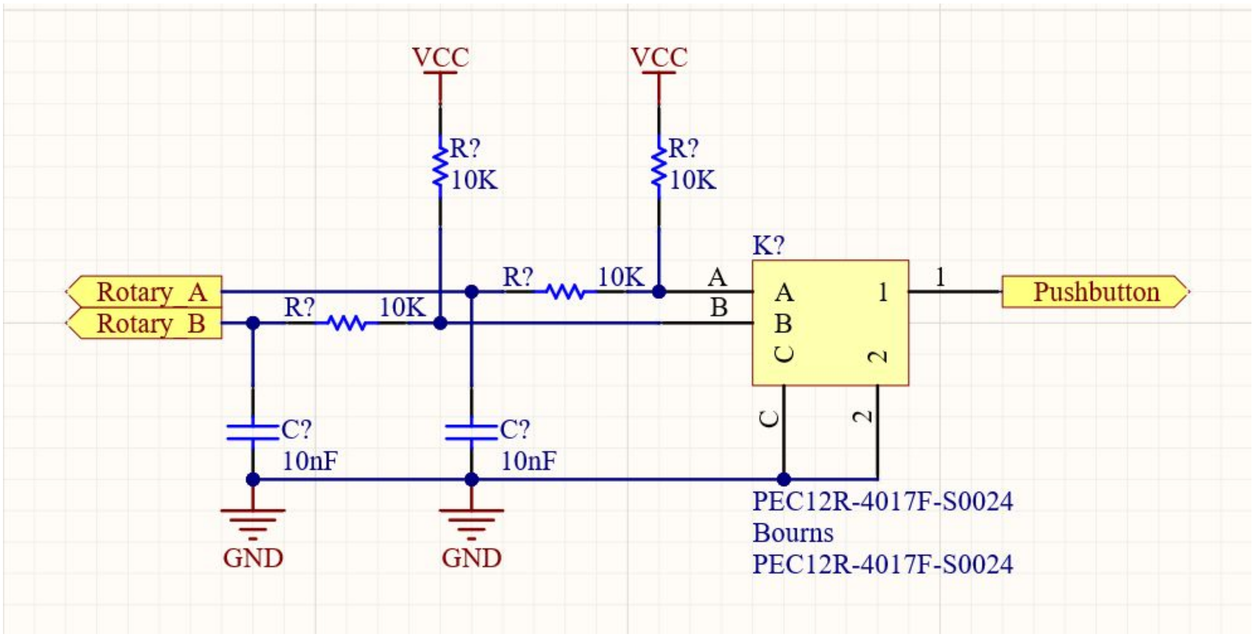
Hierarchical design makes the PCB look much neater and simpler.



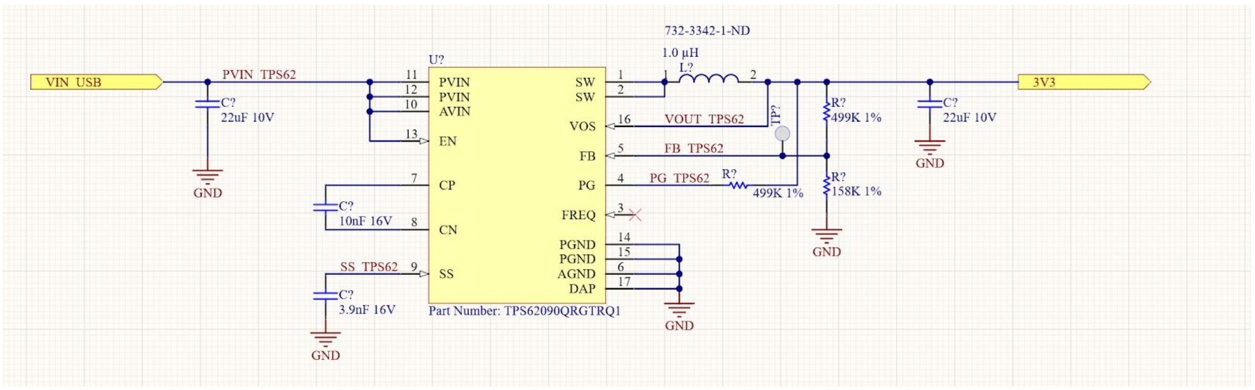
The knobs in v1 are made with Hall-effect rotary encoders. A single knob's schematic...



is duplicated neatly, so any change in one knob will automatically update to all knobs



The quadrature knobs in v2 are repeated in a similar clean pattern



Buck converter is used to regulate the power for microcontroller and peripherals