1290

UNIVERSIDADE Ð
COIMBRA

Guilherme Miguel Matos Costa

GPU Processing of 3D Audio

Dissertation in the context of the Master in Informatics Engineering, specialization in software engineering, advised by Professor Rui Pedro Paiva and Doctor Nuno Fonseca presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September of 2022

Faculty of Sciences and Technology

Department of Informatics Engineering

GPU Processing of 3D Audio

Migrate the 3D audio render engine from CPU to GPU

Guilherme Miguel Matos Costa

Dissertation in the context of the Master in Informatics Engineering, Specialization in Software Engineering advised by Dr. Nuno Fonseca and Prof. Rui Pedro Paiva and presented to the
Faculty of Sciences and Technology / Department of Informatics Engineering.

January 2022

1 2 9 0

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Đ
COIMBRA

Acknowledgements

First and foremost, I would like to thank my advisor, Nuno Fonseca, for the opportunity of being part of the software development team in Sound Particles, providing inspiration and knowledge during the full-time internship. It was a very educative and professional experience in a workspace with a group of very talented and fabulous people. I can leave the internship and say I am a more mature and developed person, and I can not forget to thank you for the experience.

I would like to thank my advisor, professor Rui Pedro Paiva who supervised this documentation and helped acknowledge all the mistakes performed while writing this document and never gave up on helping me .

I cannot also forget to thank Alexandre Frazão for supporting all my encounters during my software development and helping in all the ways he could so I could conclude my work.

I want to thank all the people in the Sound Particles company for providing an excellent experience for my first professional work in the world, full of new beautiful experiences that I would repeat.

I would like to thank my group of friends, BXD-CBR, for supporting me during the work development and helping me write and review this document.

Finally, I have also to thank my family for all the support they provided on both the internship and the writing of the thesis, where my father took his time also to revise the document so it would have fewer errors as possible.

Abstract

As society progresses, the computation power follows it so that we can achieve new ways of increasing performance in computing software. The use of the Graphics Card in software development is a perfect example because of its high thread number so that we can achieve high-performance systems.
This thesis is a part of a curricular internship with the theme "GPU 3D audio processing" for the company Sound Particles.
At the moment, the audio processing pipeline is done in Central Processing Unit (CPU), and the objective of the internship is to migrate this processing to Graphical Processing Unit (GPU) to increase scalability and rendering speed.
The development of this migration was done using the Metal API and developed in the Objective C language, which is instantiated in the language that the company's products use, C++.
Metal Application Programming Interface (API) can be invoked in Swift, Objective C, and recently with C++17 after Apple released a low-overhead C++ interface. Processing in GPU is done in Metal Shading Language (MSL), which is a low-level language, developed for API, which is based on the C++14 language, also known as ISO/IEC JTC1 /SC22/WG21 N4431.
In the first semester, the main focus was an adaptation phase where I addressed the topics GPU and General-Purpose Computing on Graphics Processing Unit (GPGPU) and developed prototypes to start development in Metal API and in Objective-C to handle all dependencies in the new development environment.
In the second semester, the main focus was completing the migration of the audio engine to GPU, followed by a phase of optimizations and performance measurement.
The render migration was completed with a speedup of 82% compared to the CPU benchmarks. Due to the prioritization of achieving a better render architecture that would present more promising results, there was not enough time to conclude it and simultaneity complete quality tests.

Keywords

General Purpose Graphics Processing Unit, Graphical Card Programming, High-performance computing, Migrate the processing from the CPU into the GPU, Metal, GPGPU

Resumo

À medida que a sociedade avança, o poder computacional acompanha para alcançar novas formas de aumentar o desempenho nos “softwares”. O uso da Placa Gráfica no desenvolvimento de “software”, devido ao seu alto número de threads, é um exemplo perfeito disso.
Esta tese faz parte do meu estágio curricular com o tema "GPU 3D audio processing" para a empresa Sound Particles.
Agora, a pipeline de processamento de áudio é feito no CPU, sendo o objetivo do estágio a migração deste processamento para GPU para aumentar a escalabilidade e velocidade de renderização.
O desenvolvimento desta migração foi feito utilizando a API "Metal" sendo instanciada na linguagem que os produtos da empresa utilizam, C++.
O Metal API pode ser invocado em Swift, Objective C, e recentemente com C++17, dado á Apple lançar uma “interface” C++ de baixo overhead, metal-cpp. O processamento em GPU é feito em MSL, sendo uma linguagem de baixo nível, desenvolvida para API Metal, baseada na linguagem C++14, também conhecida como ISO/IEC JTC1/SC22/WG21 N4431.
No primeiro semestre, o foco principal foi uma fase de adaptação onde abordei os tópicos GPU e GPGPU, e desenvolvi protótipos para iniciar o desenvolvimento em Metal API e em Objective-C para lidar com todas as dependências no novo ambiente de desenvolvimento.
No segundo semestre, o foco principal foi a migração completa do motor de áudio para GPU seguida de uma fase de otimizações e medição de desempenho.
A migração do processo de renderização foi concluído com uma aceleração de 82% em comparação com os valores obtidos pelo CPU. Foi priorizado alcançar uma arquitetura de renderização mais eficiente para apresentar melhores resulstado.No entanto, não existiu tempo para concluir esta nova arquitetura e a tarefa relacionada com os testes de qualidade não foi completada.

Palavras-Chave

Unidade de Processamento Gráfico de Propósito Geral, Programação em Placas Gráficas, Computação de Alto Desempenho, Mitigação de Processamento de CPU para GPU,Metal,GPGPU

Contents

Acronyms

API Application Programming Interface.

CPU Central Processing Unit.

DAW Digital Audio Workstation.

FLOPS Floating Point Operations per Second.

GPGPU General-Purpose Computing on Graphics Processing Unit.

GPU Graphical Processing Unit.

GUI Graphical User Interface.

HPC High-performance Computing.

MSL Metal Shading Language.

OS Operating System.

TERA 10^{12} Operations.

List of Figures

List of Tables

Chapter 1

Introduction

Presently, the importance of digital systems is a fundamental and crucial position, and society will always need to achieve more complex operations that need to be as fast and safe to utilize.
Sound Particles is an audio software company that applies computer graphics techniques to sound. Sound Particles software can deliver immersive audio software capable of generating thousands (even millions) of sounds in a virtual 3D audio world. [Sound Particles]
Each particle system can be described in a three-dimensional space and is represented by a point that can have movement or not. Each sound captured by each microphone is denoted by its gain and delay in decibels so that the system can compute the interpolation of all particle sounds and present the resulting sound.
The audio rendering engine is responsible for several jobs like computing the resulting sound at a specific frame for a given frame rate for all the particles in the system and computing the sound interpolation, calculating the position/direction of each microphone at a given frame, compute the gain and the delay value for the particle for a specific microphone for each frame, etc.
With the introduction of an audio render engine that is processed in the GPU, we can achieve a significant increase in performance due to its highly parallelized capabilities.
As such, the main objective of this work is to migrate the current cpu-based solution of the Sound Particles software to a gpu-based solution.
To construct this migration, I utilized the Application Programming Interface (API) Metal, a low-level, low-overhead hardware-accelerated 3D graphic and compute shader API created by Apple in 2014.
In order to increase the bandwidth between the CPU and the GPU, Apple designed a chip that is the CPU and the GPU together, the Apple M1 chip, , released in November of 2020 [Wikipedia contributors, 2022a]. At the time of the writing of this thesis, Apple upgraded the M1 chip, and with the new versions, the M1 Pro and the M1 Max, that resulted in a larger memory bandwidth value, more memory capabilities, and better computing power [Malcolm Owen, Oct 30, 2021].

1.1 Motivation and Scope

Today, GPU is a cheaper commodity solution to having a data-parallel co-processor alongside the CPU [Fleming, 2008] since we can acquire a cheap GPU that sill is able to deliver a good performance execute compute operations .
For many years, graphic cards have had an increasing position in the current digital world. They were responsible for displaying images and motion on computer graphics. Today, with the new APIs and the increasing developments on the General-Purpose Computing on Graphics Processing Unit (GPGPU) topic, we can almost achieve similar processing capabilities in the GPUs that until now were developed mainly in the CPUs.
Graphics cards have evolved and can be used in image processing to find similarities and differences in pixels. GPUs can process big data with their thousands of computational cores and deliver 10–100x application throughput compared to CPUs alone [Olena, Feb 8, 2018]. In deep learning development, the acceleration rate is up to 4-5 times, as revealed in some tests [Buber and Diri, 2018].
With a GPU implementation, we can accelerate algorithms that are intensive tasks. Due to the highly paralyzed capability, the GPU will present impressive results in the compute time.
To illustrate the GPU parallel efficiency, Table 1.1 shows the time it takes to add two vectors, the first row is the CPU processing time, and the second row is the GPU processing time.
The code was written in Objective-C, and the GPU API utilized was Metal, and the time recorded was after all allocations in a MacBook with a Dual Core 1.3 GHz Intel Core i5 CPU and an Intel HD Graphics 615 1536 MB GPU.

```
for(int i = 0; i < length;i++){
        result[i] = buffer[i] + buffer2[i];
}
```

Listing 1.1: CPU/Objective-C Code

```
kernel void add(const device float *arr1 [[buffer(0)]],
    const device float *arr2 [[buffer(1)]],
    device float *result [[buffer(2)]],
    uint index[[thread_position_in_grid]]){

        result[index] = arr1[index] + arr2[index];
}
```

Listing 1.2: GPU/Metal Shading Language (MSL)

The task of adding two arrays can be highly parallelized since all operations do not require elements from previous iterations or future ones, so we can divide it into a single operation and have each thread perform it.
In the CPU code, the main process iterates the same code for the length of the array we want to add, but the GPU code is run by all threads that are launched, obtaining a high-performance improvement when we increase the length of the numbers we want to process.

Array Length	CPU Time (seconds)	GPU Time (seconds)
1000	0.000397	0.000032
10000	0.005924	0.000043
100000	0.042192	0.000120
1000000	0.320999	0.001449
10000000	3.419966	0.007442

Table 1.1: Time to sum elements by the index of two arrays
Source: Author

Currently, the render engine of the Sound Particles software is a multithreaded

CPU implementation, where the producer thread launches one thread per core in the CPU and then it will divide the workload so that all threads can work in parallel.

With a new GPU integration, the engine can free some of these threads to process other work required and have a synergy between the CPU and GPU, working together.

Since the Sound Particles Software supports a live render and an action-blocking render. The architecture needs to be designed to work in the two types of render. The blocking render is the typical render button in the software, which blocks every user action until the render process is completed. Since the rendering system is blocking, we can adjust the payload(number of frames to process in a callback) to achieve better GPU render times.

The live render is more complicated to adjust the payload since the user always defines it, and it will be a small payload with a low number of frames to be computed.

That being said, the architecture needs refactoring since it processes from each particle to each microphone to a block of frames. To achieve an efficient engine, it will need to process the most particles in a payload for a block of a fixed number of frames.

1.2 Objectives and Approaches

General-purpose computing on graphics processing units is a term used for using the graphical processing unit to perform calculations instead of typical graphics rendering.

Processing data is usually a task made in the Central Processing Unit(CPU) sequentially. With a GPU computing integration, we can increase the throughput of the data processed by using a large amount of GPU cores to parallelize it if the tasks can be performed.

So far, the GPU is frequently used to process graphical data, but with the GPGPU concept, we can enhance how we process large amounts of data by separating this processing of the data into a huge amount of threads.

Currently, the audio renders pipeline of Sound Particles software uses the CPU as the main source of processing. With the integration of the GPU to the audio rendering engine, the CPU will be free to perform other tasks CPU-intensive tasks.

Presently, the Sound Particles audio engine works parallelly, where a pool of CPU threads process the data to achieve a good performance in the rendering times. Due to the complexity of the software, it is also responsible for other tasks like managing the GUI (Graphical User Interface), binaural processing case it is active, the interpolation of the air delay and mixes for the playback, etc.

To accomplish efficient software with the new implementation, the main objective of this thesis is to develop a system where the GPU makes all the major audio rendering calculations so that the CPU performs all the tasks that it previously did before and still be utilized to process new features in the software.

This implementation contains some penalties, the most important is that the GPU

requires specific commands that need to be encoded by the CPU.
Secondly, the GPU can only process data that is in the system, VRAM(Video Random Access Memory), so there is the drawback that the data need to be sent to the GPU, in order to be processed and then sent back to the CPU, which is always limited by the bandwidth of the motherboard.
Thirdly, historically GPUs are further limited in floating-point precision, with only the most recent GPUs having full 32-bit IEEE 754 single-precision floats. Modern GPUs already support double-precision floating-point operations but present a very noticeable performance drop. For example, the graphics card GeForce RTX 3090, released on September 24, 2020, has a 10^{12} Operations (TERA)Floating Point Operations per Second (FLOPS) capability from 29.28 to 35.58 for single-precision operations, and a TERAFLOPS power from 0.459 to 0.558 to double-precision operations, which present a speedup of 63.7 from single-precision to double-precision operations.

1.3 Results, Contributions, and Limitations of the thesis

The work developed in Sound Particles can be divided into two prototypes, the first one that is just a complete migration of the current engine but performs all calculations on the GPU step by step, and the second prototype being a more efficient prototype that utilizes larger kernels so that it does not exist an overhead.
During the migration of the Sound Particles Objective-C bridge to C++ to the Apple interface, the developer Alexandre Frazão worked alongside me. He implemented a bridge between the interface objects and the C++ since the interface did not contain an autorelease feature and utilized the C++ garbage collector to destroy all objects that were not used during the application runtime.
The first prototype principle was utilizing small kernels to perform minor operations required by the engine, like processing particles and microphone positions followed by the polar response, etc. This resulted in overhead from constantly switching from kernel to kernel, affecting the system's efficiency.
To eliminate this overhead, a second prototype was developed with the intuitive to replace the first and be a conjunction of all small kernels developed in the first prototype. Due to a lack of time, this prototype was not completed, and I could not obtain the benchmark results. That being said, this thesis will revolve around the first prototype and all the results that revolved around it.

1.4 Sound Particles software

1.4.1 What is a Particle System

In order to describe the Sound Particles software, an explanation of a *particle system* in the sound industry need to be defined. In a system where we have one or more groups of particles, and each one will emit a sound, the microphones in the system will capture its sounds, and by counting the speed of sound, the distance, and all the natural world limitations will reproduce a final sound result.
The particles in these groups can have a vast number of shapes, and the microphones can also be of different types, ambisonics, multichannel, etc..
It is intuitive that to achieve the output sound, we will need to do a tremendous number of computations.
Currently, the render engine can be simplified to the following steps:

1. Compute particle position
2. Compute microphone position
3. Compute microphone direction
4. Compute pitch change (if changed)
5. Compute polar response for that particle and that microphone
6. Compute attenuation for the polar response
7. Compute sound interpolation

1.4.2 Sound Particles

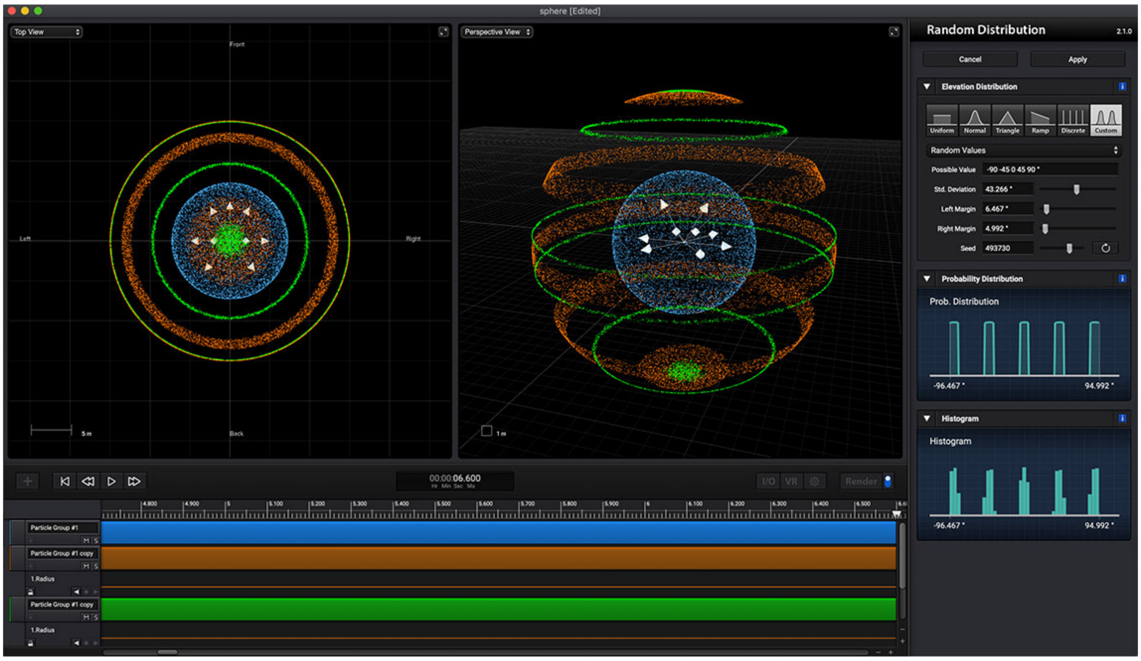
Sound Particles 2

“Sound Particles” is an audio software company that applies computer graphics techniques to sound. The software is utilized in top videogame companies (Blizzard, Epic Games, PlayStation) and all major Hollywood studios, being used in productions such as “Game of Thrones,” “Frozen 2”, “StarWars 9”.” [Sound Particles]
The software uses computer graphics and visual effects techniques within the audio segment. It can be compared to an image edition software like Photoshop, which is suitable for editing images in 2D, but there are tools like Maya and Blender that allows edition images within the three-dimensional environment.
Currently, Sound Particles leading audio software in the market is Sound Particles 2, where the user can create sounds in virtually any output format to achieve the desired level of immersion.
It is a Digital Audio Workstation (DAW) introduced in 2016, it is designed to offer

unique and efficient sound design workflows, increase creativity and reduce the time needed to create and record complex sound effects in virtual 3D environments.

The software can simulate any output virtually format possible, providing microphones systems of the following types: mono, stereo, ambisonics, 5.1, 7.1, Dolby Atmos Beds, Auro 3D, and many more. It also supports batch processing, CGI and video integration, and binaural monitoring and simulates the doppler effect if the user desires.

If the user wants to change the system along the time, it is also possible to randomly change the equalizer, the sound pitch, the sound delay, the granular sound, and particle group movement by creating velocity or acceleration, ... The Figure



Random Values

Possible Value -90 -45 0 45 90 °

Std. Deviation 43.266 °

Left Margin 6.467 °

Right Margin 4.992 °

Seed 493730

Figure 1.1: Sound Particles 2

1.1, we can observe the software with a standard sample template that simulates multiple spheres composed of tiny particles that radiates sound that is then captured by a microphone, that by its nature will receive different types of results. The microphone can be a traditional stereo or a more complex one like a binaural microphone within all the possibilities of microphone types provided in the software.

The software also presents other features like allowing the sound engineer to add movement modifiers to the particles, simulating the sensation of a moving sound, among other features.

Initially, the software allowed the power to create immersive sounds with simple clicks on a button, which evolved into complex sound software that allows the users to deliver sounds it was not possible before with all the new functionalities.

Sound Particles 3

The Sound Particles 3 is still in development, and by the time this thesis got written, it did not get the alpha version released to the public/alpha-testers.
It will revolutionize the Sound Particles 2 interface adding a better graphic user interface to the user and presenting a bunch of new features, but the most important to this thesis is the opportunity to change the render system to the GPU, among other features that are not important to this document.
That being said, the primary purpose of this work is to integrate this version of Sound Particles by completing the migrating of the render engine to the GPU. So that we can achieve better-reduced render times than the CPU ones.
The work developed was a complete migration of the render engine to the GPU, but since the results were around 80% of speedup, it was not enough, so a complete architecture refactoring was needed.
To develop a new render engine, refactoring and tests to verify all results were needed. These tests need to have a margin of error since the render engine in the CPU utilizes double as the primary data type, and the Metal API only can utilize a single precision storage format, like float and half.

1.5 Outline of the thesis

The first chapter of this thesis summarizes the work produced, all results achieved with the work developed, and a small presentation of the company Sound Particles and all their products by the time this thesis was written.
The second chapter resolves around the planning of the work that would be done, followed by the description of the development process and finish with the risks associated with the work.
The third chapter settles about GPGPU describing it, why it should be used in current software to achieve better performance in applications, and all the APIs that can be used to develop general-purpose computing on graphics processing units.
The fourth chapter is a description of the requirements and the architecture implemented in the prototype, containing the Sound Particles user stories that were written before my internship started and some that I added.
The fifth chapter explains the implementation to achieve the solution to the problem proposed in this thesis, providing a detailed perspective on the solution achieved.
The sixth chapter is a presentation of the results achieved with a complete description of how the benchmark tests were made and all the quality tests developed until the moment.
In the seventh chapter, the final chapter, it described a conclusion of the work developed and all the future work that can be implemented to achieve a more efficient system.

Chapter 2

Project Planning and Management

2.1 Planning

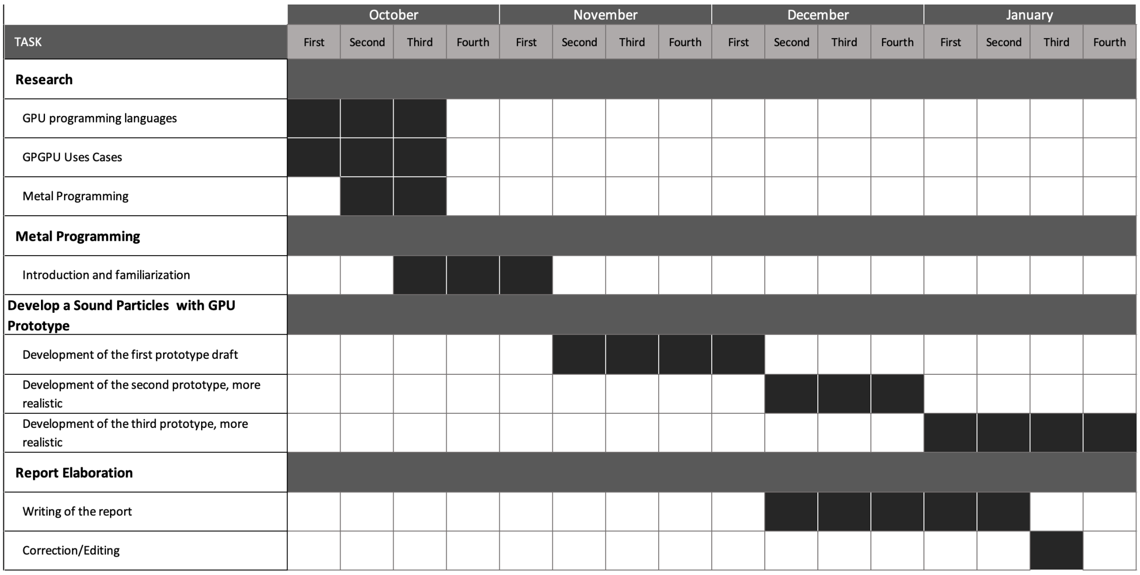
The internship started on October 8, 2021, with my visit to the company's main facilities in Leiria, where my advisor Nuno Fonseca presented their software and the Sound Particles team.
The following weeks, until the beginning of November, were a phase of research on current technologies and how I use them to implement the engine migration. During this time, I chose the API that I will utilize during the migration, and I started making some initial development with the API. The first experiment utilized the chosen API to add all elements of two arrays into a new array.
After this initial testing, the first step was implementing a prototype with some principles that the Sound Particles Software utilizes, like calculating the length between two points and developing multithreading to divide the workload between threads since the workloads are pretty heavy.
After this initial prototype, the following work was on a second prototype, which involves broader operations that occur in the current engine, for example, to calculate the gain and the delay of each particle (per particle and per microphone way). However, with the increased workload, some memory difficulties were encountered. At this stage, there were some divergences between my approach to the problem and the tutor vision when I performed more intensive workloads to the GPU but were not scalable, which also slowed down this development.
The workload was composed of 10 particle groups composed of 1.000.00 particles each, and compute the gain and the delay to 10 microphones for 60 seconds with a sample rate of 48 kHz.
During the first semester, the plan was to perform a third prototype that would be even closer in operations to the current engine. However, the second prototype memory management slowed my progression. Therefore, this objective was not attained.
The full-time work started on February 15, 2022, wherein the first weeks were to set up the workspace and study all the Sound Particles modules, specifically the sp-engine and the sp-core described in the System Desing chapter, the fourth chapter. The following weeks and months were to develop the render migration to the GPU. When I reached the final part of the render engine, a new phase of

optimizations started since it did not deliver the expected results. In the beginning, there were just small CPU and GPU code optimizations, and it still did not beat the CPU render duration. So a phase of refactoring the interpolation process resulted in a GPU render time improvement over the CPU of almost 20%. While this task was being developed, a testing phase also started where I developed functional and quality tests for the new engine.
To achieve the desired results of 80% improvement, a new render engine architecture was required, so on August 8, I started developing it.
This task started with the development of new compute shaders that were a concatenation of the previous ones, developed in the first migration, and after it, they needed a refactoring process since each shader was made to process a single particle and to achieve the desired results it needed to process a batch of particles for all microphones in a microphone system.
This task was not completed even with the shaders code finalized, the sp-engine code not getting concluded to utilize this new architecture, but the most part was developed.
The following tasks describe the initial planning of the internship:

1. First Semester:
   (a) Understanding Sound Particles (knowing the software, what it does, its internal structure, the building pipeline, etc.)
   (b) Understanding the GPU architecture, their development process, and their tools.
   (c) Design and implement a Sound Particles GPU integration architecture
   (d) Start migrating audio rendering code from CPU to GPU.
   (e) Write the mid-project report
2. Second Semester:
   (a) Full migration of the audio rendering code into GPU.
   (b) Performance measurement and optimization
   (c) Documentation and final report

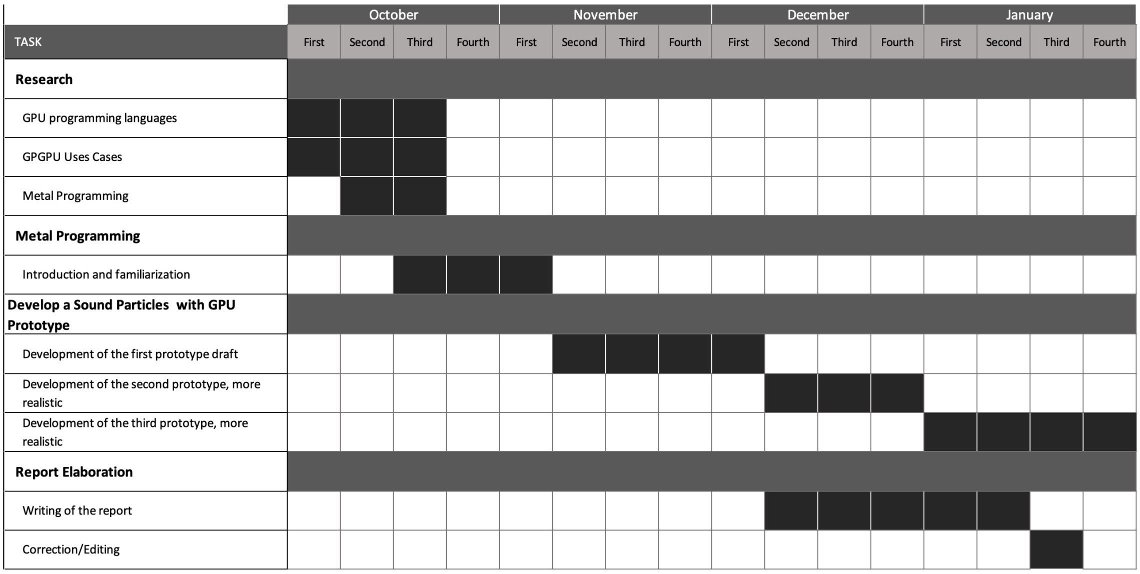
Since it this initial purpose has information about the average time it took to perform the tasks. The following diagram presents my interpretation of the supposed planning of the internship/thesis.
We can observe Figure 2.2 that the development of an actual prototype took a very long time to develop compared to the expected, Figure 2.1, this is caused because of the heavy conditions that the prototype was submitted, like a sample rate of 48000 with 1000 particles per group in ten groups with ten microphones, over 60 seconds.
To develop this robust prototype so that it could handle this amount of workload, which will need even improve more during the development of the internship, still represents a small prototype compared to the current Sound Particles software.
The GANTT diagram in the Figure 2.4, presents my planning for the second

semester, where I will implement the full migration of the audio rendering engine to the GPU followed by a phase of testing and a phase of making all optimizations that were not made during the development and finishing with the writing of the report.



	October				November				December				January			
TASK	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth
Research																
GPU programming languages	■	■	■													
GPGPU Uses Cases	■	■	■													
Metal Programming		■	■													
Metal Programming																
Introduction and familiarization			■	■	■											
Develop a Sound Particles with GPU Prototype																
Development of the first prototype draft						■	■	■	■							
Development of the second prototype, more realistic										■	■	■				
Development of the third prototype, more realistic													■	■	■	■
Report Elaboration																
Writing of the report										■	■	■	■	■		
Correction/Editing															■	

Figure 2.1: Planning for the First Semester



	October				November				December				January			
TASK	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth
Research																
GPU programming languages	■	■	■													
GPGPU Uses Cases	■	■	■													
Metal Programming		■	■													
Metal Programming																
Introduction and familiarization			■	■	■											
Develop a Sound Particles with GPU Prototype																
Development of the first prototype draft						■	■	■	■							
Development of the second prototype, more realistic										■	■	■				
Development of the third prototype, more realistic													■	■	■	■
Report Elaboration																
Writing of the report										■	■	■	■	■		
Correction/Editing															■	

Figure 2.2: Actual development for the First Semester

	February				March				Abril				May				June				July			
TASK	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth
Full migration of the audio rendering code into GPU																								
Setup of the work environment	■																							
Mitigate CPU processing to GPU		■	■	■	■	■	■	■	■	■	■	■												
Testing						■	■	■	■	■	■	■	■	■										
Result evaluation															■									
Performance measurement and optimization																								
Compare statistics of processing															■									
Make Optimizations																■	■	■						
Documentation and final report																								
Writting of the document															■	■	■	■	■	■				
Editing and adjutments																				■	■			

Figure 2.3: Planning for the Second Semester

	February		March				Abril				May				June				July				August			
TASK	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth	First	Second	Third	Fourth
Full migration of the audio rendering code into GPU																										
Setup of the work environment	■																									
Study of the Sound Particles Modules	■	■																								
Migrate the render engine to the GPU		■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■			
Develop the new arquitecture																							■	■	■	■
Develop GPU render tests																					■	■				
Performance measurement and optimization																										
Compare render statistics with the CPU					■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■	■
Make Optimizations																	■	■	■	■	■	■	■	■	■	■
Documentation and final report																										
Writting of the document																	■	■	■	■	■	■	■	■	■	■
Editing and adjutments																				■	■	■	■	■	■	■

Figure 2.4: Actual development in the second semester

2.2 Development Process

2.2.1 First Prototype

This prototype developed was very simple in terms of complexity since the main objective was to help me to understand the development of code with the Metal API.
This prototype objective was to process small amounts of data in the GPU and perform small and the main operations made in the Sound Particles Software, like subtracting and adding values and calculating the length between two 3D points.

2.2.2 Second Prototype

The second prototype was a Mini Prototype of 3D particle render, where there were the concepts of groups of particles and microphones present for each particle and each microphone at a given time step, the polar response, the gain, and the delay.

Also, to develop an efficient prototype, I developed in this stage a metric system with the intuitive to present if all the improvements developed during the prototype were beneficial or not.

2.2.3 Third Prototype

The third prototype is an even more complex prototype that is meant to be as similar as possible to the current Sound Particles 3D particle render engine, making all the calculations that the actual renderer performs.

2.2.4 First Sound Particles Prototype

Since February 15, 2022, the main focus of the work has been to migrate the render engine to the GPU, which was achieved on July 23, 2022. Since the speedup achieved was 82.00%, it was not enough to justify all work developed, so a new approach was needed.

This prototype can be decomposed into a pack of requests to process sound emitted by the particles for the microphones in the system. This pack comprises a request for particle position, microphone position, polar response for each particle for each microphone, sound attenuation, and sound interpolation. To process each particle and for each microphone for a sample rate, we need five requests to the GPU, which can create an overhead between the CPU and GPU.

2.2.5 Second Sound Particles Prototype

The development of this prototype was a follow-up to the first one. It just picks up all the work developed during the render engine and compresses it in only one request instead of an abundance of them.

That being said, the main focus of this prototype is to create kernels that capsulize all the small requests into just one big one so that we do not have all the unnecessary overhead.

This new architecture consists of deploying threads that follow the number of particles instead of, like in the first prototype, just launching the number of threads that were needed for each particle and each microphone. This work implied a significant change in the render architecture since it was not developed for this render path.

2.3 Risks

The implementation of this assignment comes with some risks I focused on the first semester to overcome when I started to migrate the processing to the GPU. The Following list demonstrates the main risks that were raised in the first semester of adaptation:

1. GPGPU Programming
   (a) No prior experience programming GPGPUs. My only experience was in the topic of shaders in the Computer Graphics course of the Bachelor Program on Informatics Engineering, where we programmed texture shaders in OpenGL.
   (b) Mitigation plan: Develop prototypes that improve the complexity over time to improve my abilities
   (c) Probability of occurrence: Average
   (d) Impact: High
2. Large Amounts of Data to Process:
   (a) A large amount of data to be processed must be computed with high precision to deliver high-fidelity result sound with an immense immersion so that any error will result in serious problems.
   (b) Mitigation plan: Heavy testing after the conclusion and debugging tools to verify the occurrence of UB(Undefined Behaviour) in the application
   (c) Probability of occurrence: High
   (d) Impact: High

Chapter 3

Background Check and State of the Art

This state of the art resolves around the GPGPU since the main function that will perform is the migration of data processing from the CPU to the GPU.

Historically, the earliest GPUs were hard to program for anything other than graphics applications because every operation had to be mapped to an equivalent graphics operation.

In 1987, the British mathematician John Horton Conway became one of the first examples of general-purpose computing. He developed a cellular automaton using an early stream processor called a blitter to invoke a special sequence of logical operations on bit vectors [Wikipedia contributors, 2022b].

With programmable shaders and floating-point support on graphics processors, this topic became a more popular topic since the graphical cards became more important and programmable .

In 2001 the E. Scott Larsen and David McAllister performed the "Fast Matrix Multiplies using Graphics Hardware" article that showed an implementation of matrix multiplications performed in the GPU and demonstrated the amount of speedup resulting with it [Larsen and McAllister, 2001].

3.1 Central Processing Unit (CPU)

So far, to process all kinds of data, we mainly utilize the CPU to achieve fast processing by utilizing its powerful cores, but what is a CPU?

A Central Processing Unit, also known as CPU, is the brain of a computer that processes all instructions that need to be processed. They can be arithmetic, logic, and I/O operations specified by a software program.

The CPU can be decomposed as an arithmetic-logical unit(ALU) that performs all arithmetic and logical operations, the process register that supplies the operants for the ALU and then stores its result, and the control unit that is responsible for fetching memory, decoding, and executing the instructions by coordinating the operations to the ALU, register and other components.

Nowadays, the vast majority of CPUs are multicore processors, which means that on one single integrated unit, we will have two or more separate processing units(CPU), so these days the CPU that is composed of multiple separated CPUs that communicate by the bus interface and L1, L2 and the new L3 cache.
With the new multicore processors, we can achieve parallel computing, and if the cores also support multithreading, we can improve it even more, as we will be able to create additional virtual or logical CPUs.

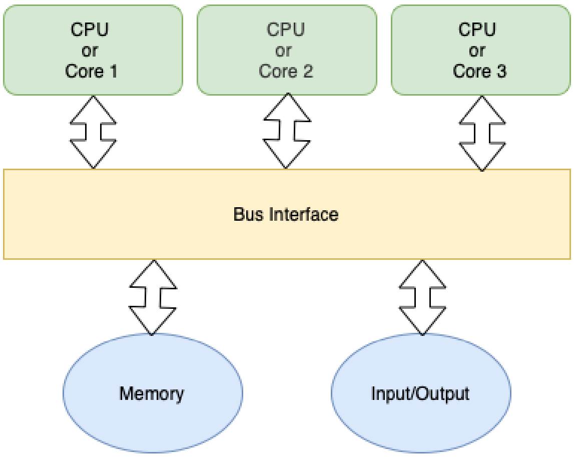


Figure 3.1: Simplified Architecture of a Multi-Core CPU
Source:Author

3.2 Graphical Processing Unit (GPU)

Modern computers come with a Graphical Processing Unit, also known as the GPU is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for the display device.
Modern GPUs effectively manipulate computer graphics, image processing, and other computing operations by utilizing their highly parallel structure.
This structure composes of numerous less powerful cores than the GPU, but their purpose is to divide the data into chunks, and then each core processes small chunks of data.
Their initial design was to accelerate the rendering of 3D graphics, but over time they became more flexible and programmable, enhancing their capabilities to dramatically accelerate additional workloads in high-performance computing, deep learning, neural networks, simulations, and more kinds of data processing, also known as General Purpose Graphics Processing Unit(GPGPU).
A single GPU device consists of multiple Processor Clusters (PC) that contain multiple Streaming Multiprocessors (SM). Each SM accommodates a layer-1 instruction cache layer with its associated cores.
Typically, one SM uses a dedicated layer-1 cache and a shared layer-2 cache before pulling data from global GDDR-5 (or GDDR-6 in newer GPU models) memory. Its architecture is tolerant of memory latency.

Compared to a CPU, a GPU works with fewer and relatively small memory cache layers.

The reason being is that a GPU has more transistors dedicated to computation, meaning it cares less how long it takes the retrieve data from memory. The potential memory access 'latency' is masked as long as the GPU has enough computations at hand, keeping it busy the potential memory access 'latency' is masked as long as the GPU has enough computations at hand, keeping it busy [VMWare, 2022].

NVIDIA realized the potential of bringing this performance to the larger scientific community and invested in modifying the GPU to make it fully programmable for scientific applications.

The typical architecture of an Nvidia GPUs is illustrated in the figure 3.2 with a description of all its components.

3.2.1 GPU Programming Limitations

Despite the enormous performance of the GPU, it comes with some drawbacks because they still are simple operations translated that were encoded by the CPU to be executed by it.

It does not have virtual memory support, privilege levels, I/O support, and other facilities that a CPU provides for operating a general computer. The main disadvantages can be observed in the following list:

1. GPUs are much harder to program than CPUs due to his different and restrictive programming model

2. Different workflow between GPU APIs

3. GPU API devices and/or operative systems limitations

4. The data must be repositioned to the GPU to be processed and then sent back to the CPU, which limits the process

5. Some of the shading languages do not support double-precision numbers, and the ones that support them execute much slower. For example, the NVIDIA RTX 3090 in single precision operations has a performance of 35.58 TFLOPS, and in double precision, operations perform at 556.0 GFLOPS.

6. The processing of the GPU must be encoded by the CPU, so for an efficient system, we must have multiple threads to encode.

7. Kernel errors are very hard to debug since we cannot check variables values at a running time like in the CPU

8. Memory Management can be tricky to implement and have lower memories than the CPU

9. Programming in GPU is still an evolving technology

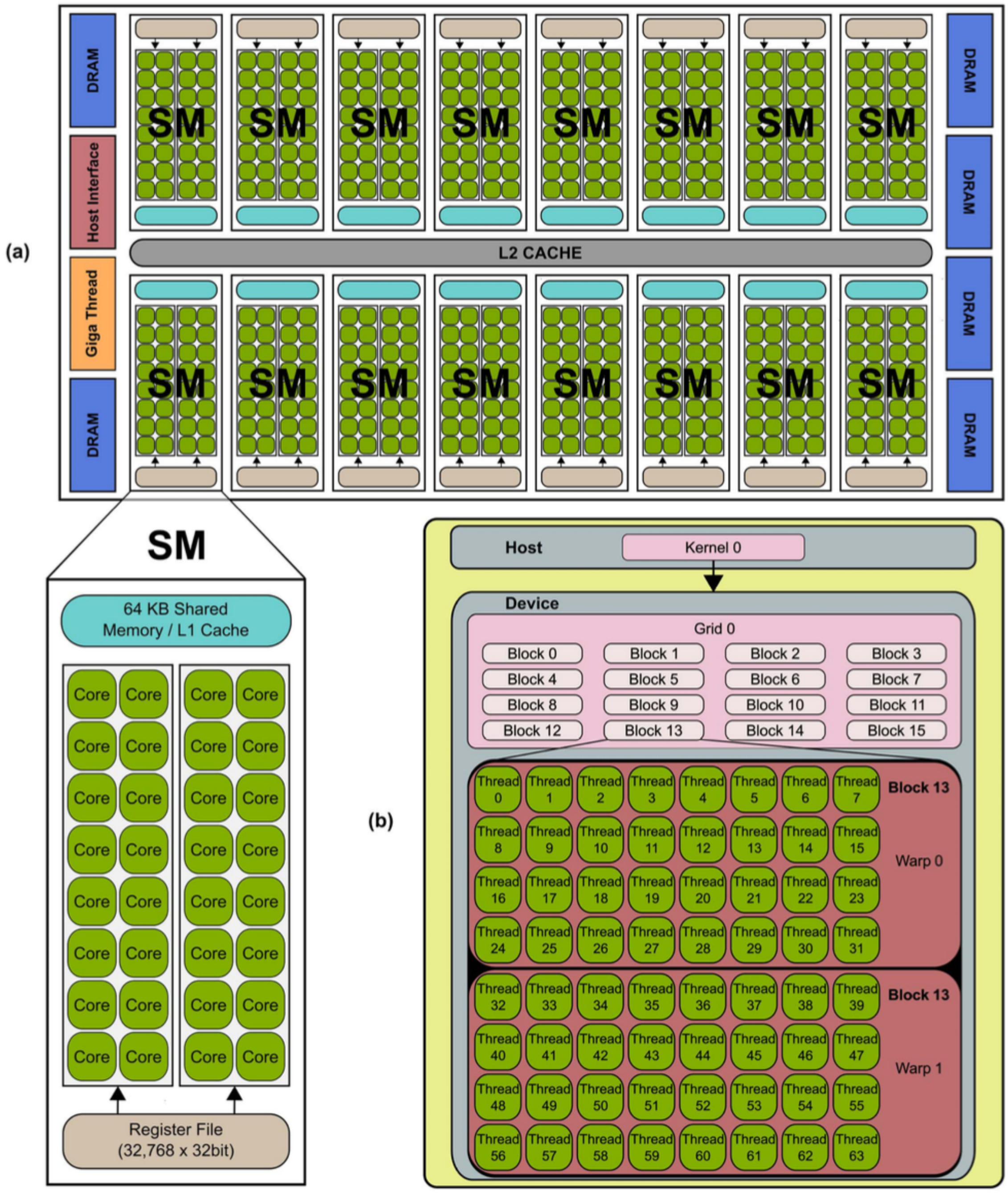


Figure 3.2: Typical NVIDIA GPU architecture
(a) NVIDIA's Fermi architecture.
(b)The GPU resources are controlled by the programmer through the CUDA programming model.

Source: [Hernandez M, 2013]

10. While GPGPU can achieve a 100-250x speedup vs. a single CPU, only embarrassingly parallel applications can develop these speedups

11. Nvidia and AMD offer to compete for GPU programming languages that only work with their products

3.3 GPGPU

3.3.1 What is GPGPU

GPGPU is becoming an increasingly viable option for acceleration, including in the audio domain since we have many operations in the background.
The GPU work required mapping of scientific code to matrix operations to manipulate triangles, and it would require an insane passion and commitment to using the GPU for another purpose other than rendering graphics. Nowadays, we have wide options of APIs to choose from, enabling programmers to utilize the massively parallel power of the GPU for purposes other than graphic rendering.

3.3.2 Why use the GPU

Given the GPU parallelism architecture, a code with large loops, especially nested loops, can be simplified to a single GPU run by partitioning the operations used into independent tasks that are going to be executed by the blocks of threads. If we can partition these operations even more, we can execute them cooperatively in parallel by the threads within the block enabling automatic scalability.

3.3.3 GPGPU in Modern DAW

To express the speedup that could be achieved in a DAW utilizing a gpu render system, the following subsections would represent the speedup that can be accomplished with the change of the render engine device.
Therefore, it will only present the ones with a GPU acceleration for the render engine, but since there aren't any DAW's that support natively a GPU render engine in the macOS operative system I could not present the results. I tested the following DAW's to verify if there was a path to only utilize the GPU, but there were none. I tested the following :

1. FL Studio
2. Apple Logic Pro
3. Steinberg Cubase
4. PreSonus Studio One
5. Bitwig Studio
6. Apple GarageBand
7. Avid Pro Tools

By testing different DAW's, we can verify that there is no implementation of a fully gpu render engine, and even Apple's software, Logic Pro or GarageBand utilize it.
Accordingly, we can verify that the complexity of the problem being proposed around this thesis is very high.

3.3.4 High-performance Computing (HPC) on the GPU

The HPC term is the ability to process data and complex data at high speeds. Normally this term is associated with having a distributed network of computers that compute data together and using a computer as a cluster. However, we can also utilize the GPU as the primary computing cluster to compute data.
The Graphical Processing Unit has evolved from a graphical unit into many other purposes, like high-performance computing(HPC). Since the GPUs originally got designed to perform the image rendering that is achieved by arithmetic floating-point operations, we can achieve a similar behavior to process floating-point computations.
Modern GPUs contain hundreds of processing units capable of computing enormous amounts of data at a time, and some of this computing power is not translucent as it should be.
The new PS5 for example is capable of doing 10.28 TERAFLOPS of data, which means that it can compute $10.28 * 10^{12}$ floating-point operations per second. The main purpose of a PS5 still is to render computing graphics, but we can observe that we have tremendous power inside it.

3.3.5 CPU vs GPU

The major difference between the CPU and the GPU is the highly paralyzed architecture of the GPU that makes it much more efficient than the CPU to process data that can be partitioned and processed in parallel.
The GPU design is more focused on processing data rather than data caching and flow control, so if we have a problem that can be decomposed and processed in a parallel way, it usually means that, or the same operation is made into every element which means that we do not need a complex flow control, or the data is massive and high on arithmetic operations which reduces the need for low latency memory(cache).
An important factor between CPU and the GPU is the quality of the output after the render process since CPUs have fewer cores when compared to GPUs they are far more versatile and design to carry out complex instructions sets that allow the CPU to run almost any algorithm with little effort and provide a better quality result. This better quality is a response to the speed that the GPU provides due to modern APIs providing much better results utilizing a single precision storage format which fails in providing accuracy in the results.
Nowadays, we have a wide variety of applications with one or more elements

that can be decomposed in a parallel manner, so to achieve high efficiency, we can use the GPU to perform all the large data arithmetic operations. For example, blockchain mining was firstly developed on the CPU, but with its limited processing speed and high power consumption, it was a very inefficient process. With the entry of the GPU into mining, we attended an improvement by 800 times compared to the CPU mining.[SHOBHIT SETH, reviewed by ERIKA RASURE, updated at 25 of August in 2021]

We can observe in the figures 3.3 and 3.4 that in terms of performance, GPU surpasses the CPU in all terms except for the Intel Xenon Phi Processor that is mainly utilized in server workstations since its price is around 12 404,87 euros, compared the high-end AMD Firepro W9100 graphics card that is around 5 000 euros.

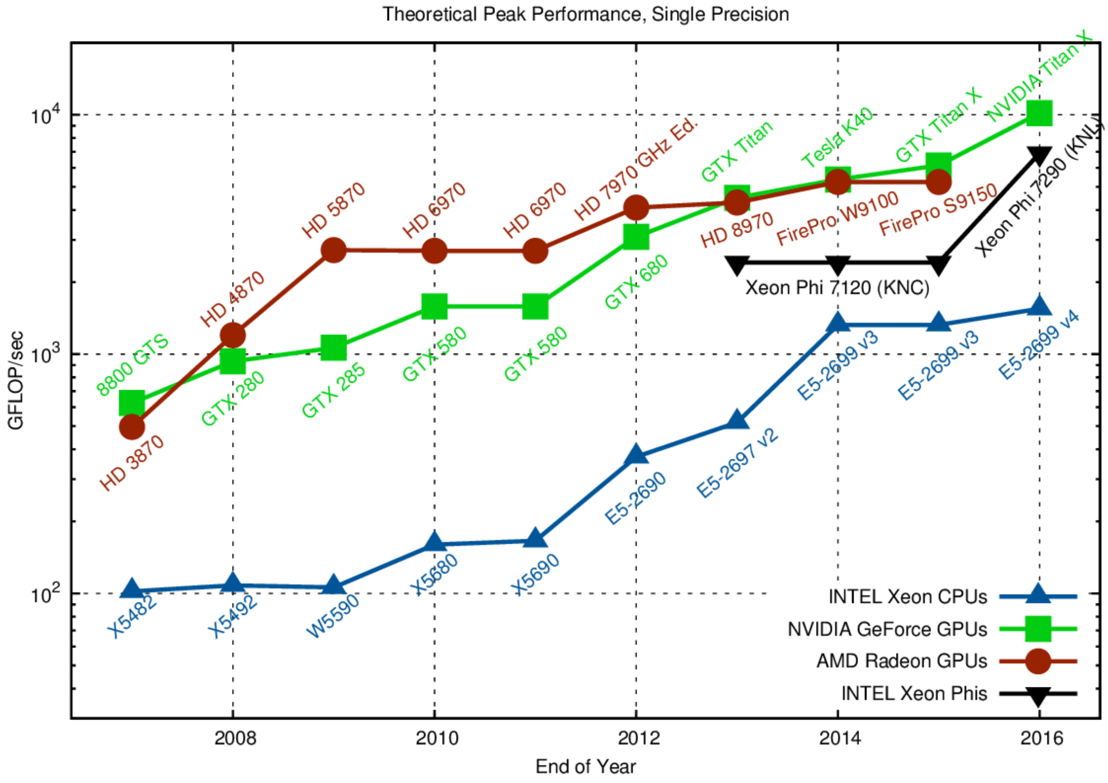


Figure 3.3: Theoretical Peak Performance for Single Precision Operations
Source: [Karl Rupp, Edit, August 18th, 2016]

We can also observe that the high-end graphics cards compared to the high-end CPUs provides better computing power, but there is still the bandwidth problem that we can observe in the figure 3.5 that, and observe that the powerful processor Intel Xenon Phis has a similar bandwidth of the presented graphics cards.

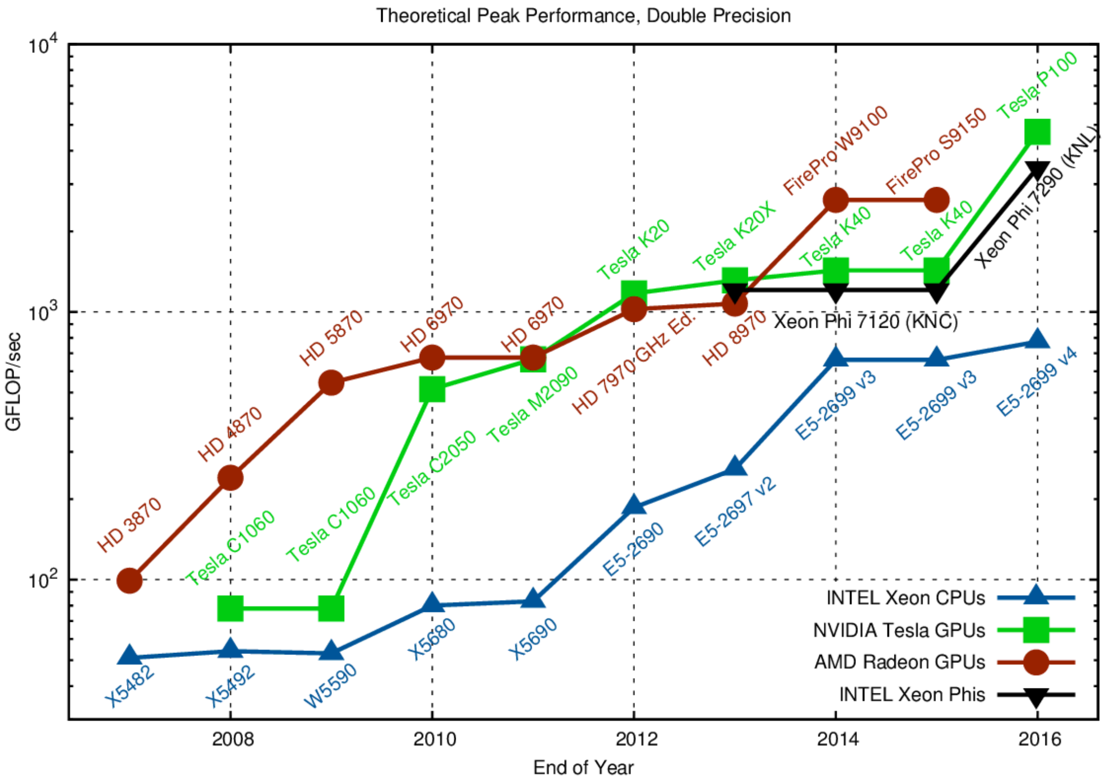


Figure 3.4: Theoretical Peak Performance for Double Precision Operations
Source: [Karl Rupp, Edit, August 18th, 2016]

3.3.6 Applications for GPGPU

Neural Networks

As real-life problems grow increasingly complex and demanding, parallel implementations of Machine Learning algorithms become crucial for developing intelligent real-world applications. In this context, the GPU is particularly well-positioned to fulfill this need, given its availability, high performance, and relatively low cost.[Lopes and Ribeiro, 2011]
Until now, most machine learning work is mainly done by the CPU. However, with new research and new implementations, we can verify more satisfying results by being more efficient due to its massive parallelism potential, by mitigating this processing to the GPU, since most APIs are also enabling a more straightforward way to develop this kind of work in the Graphical Processing Unit, like NVIDIA CUDA-X AI, Kompute Framework and Vulkan SDK.
The article [Fonseca and Cabral, 2017] presented the benchmarks of creating a neural network in the GPU, providing then the benchmarks of the CPU for the same neural network to understand better the benefits of migrating to the GPU.
We can observe in the image 3.6 that the GPU version presents better performances and a narrower sample of results that are almost constant in the 30 iterations conducted. So we can conclude that in the Neural Networks topic, the GPU is a must-have since it presents simultaneity, better performance, and more stable learning rendering results.

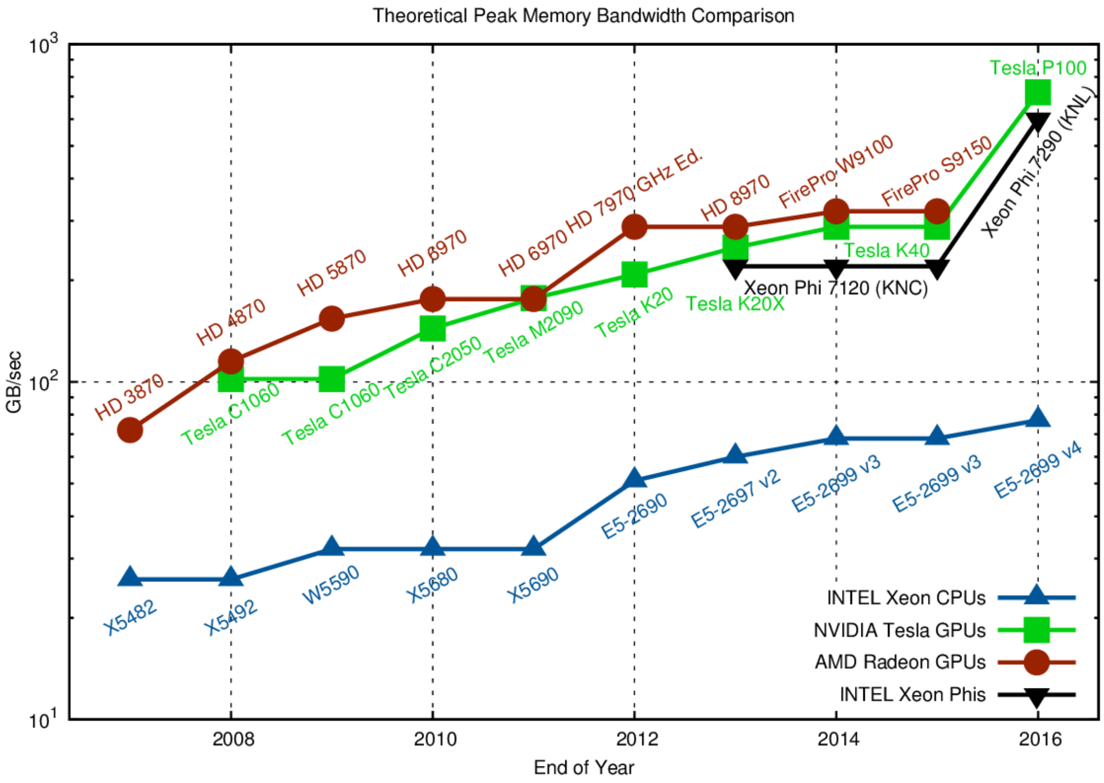


Figure 3.5: Theoretical Peak Memory Bandwidth
Source: [Karl Rupp, Edit, August 18th, 2016]

The article also presented that the GPU version executes in one-fifth of the time of the CPU, even if written in a high language like python and runs on a singleGPU, so if they performed this experiment in a lower language like C or C++ and on more devices, it would have presented even better results.
A real case of this implementation is the Tesla company, where the per-camera networks analyze raw images to perform semantic segmentation, object detection, and monocular depth estimation.
Their birds-eye-view networks take video from all cameras to output the road layout, static infrastructure, and 3D objects directly in the top-down view.
Its networks learn from the most complicated and diverse scenarios in the world, iteratively sourced from their fleet of nearly 1M vehicles in real-time.
A full build of Autopilot neural networks involves 48 networks that take 70,000 GPU hours to train.[Telsa]

Geometric Computing

Computational geometry has excellent applications in computer graphics, computer-aided design, visualization, scientific simulation, etc. [Qi et al., 2019]
In the study [Qi et al., 2019], we can observe that in the figure 3.7, we obtained better performances, compared with the CGAL and Triangle software, in all 3D models.
Their experiment was to construct a 2D, 3D Delaunay triangulation and 2D con-

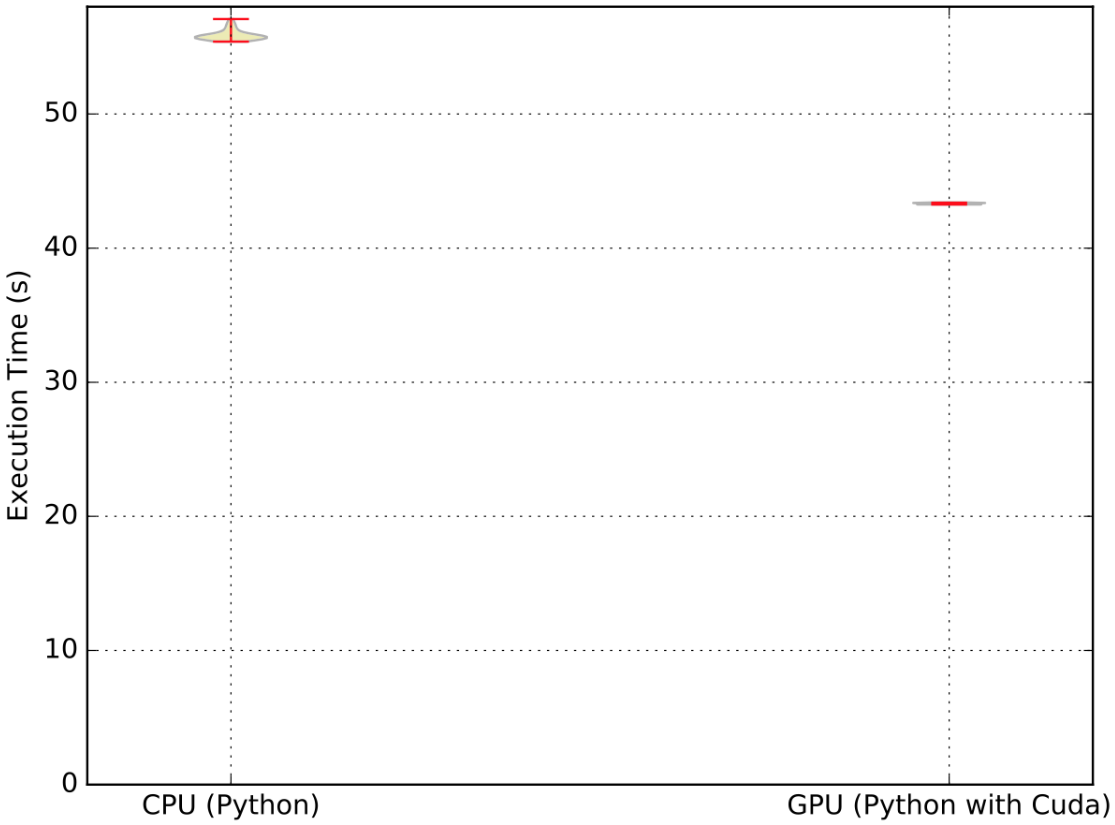


Figure 3.6: Comparison of the execution times between CPU and GPU versions
Source: [Fonseca and Cabral, 2017]

strained Delaunay triangulation for both synthetic and real-world datasets. A GPU technique can speed up the computing time for predicates by 3 to 4 times, which we can observe in the Figure 3.14, and can be used as the GPU version of Shewchuk's work for other computational geometry problems [Qi et al., 2019].

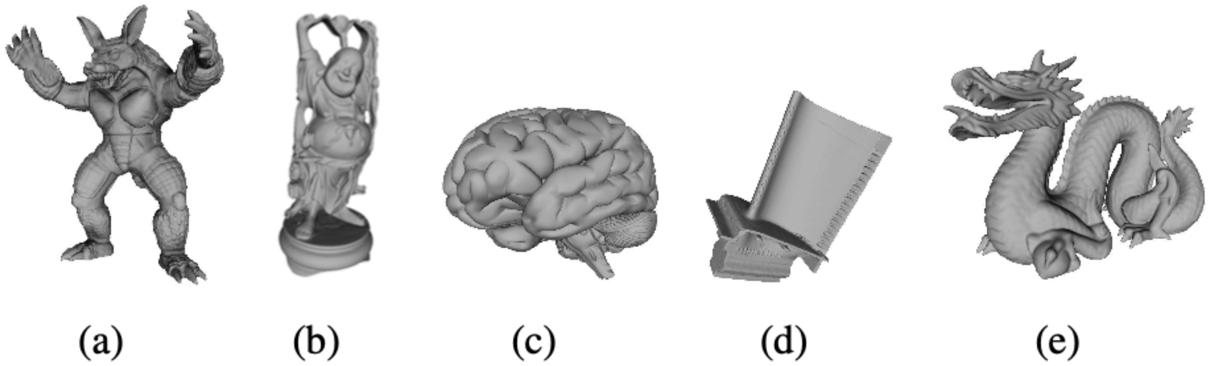
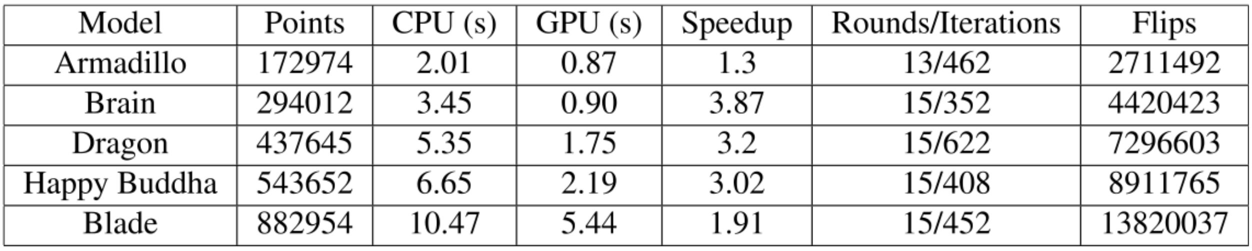


Figure 3.7: 3D models used in experiments. (a) Armadillo. (b) Happy Buddha. (c) Brain. (d) Blade. (e) Dragon
Source: [Qi et al., 2019]



Model	Points	CPU (s)	GPU (s)	Speedup	Rounds/Iterations	Flips
Armadillo	172974	2.01	0.87	1.3	13/462	2711492
Brain	294012	3.45	0.90	3.87	15/352	4420423
Dragon	437645	5.35	1.75	3.2	15/622	7296603
Happy Buddha	543652	6.65	2.19	3.02	15/408	8911765
Blade	882954	10.47	5.44	1.91	15/452	13820037

Figure 3.8: Statistics for real-world dataset
Source: [Qi et al., 2019]

Mathematics and Scientific Computing

Currently, GPUs have emerged as the dominant data-parallel coprocessor by achieving the highest performance (in terms of Tflops) since in scientific data processing needs to process a large number of numbers, the GPU can improve the performance of these challenges and accomplish outstanding performance values.
Many tests see a speedup compared to the CPU implementation, but they state their main bottleneck is the reduction check when performing a widening or emptiness test operation. They state this bottleneck could be alleviated with random writes, which (scatter) can be done with the vertex processor or with a GPGPU language such as CUDA or CAL.[Fleming, 2008]
For example, a pseudo-random numbers generator, a GPU implementation revealed, on average, a 26x speedup compared to CPU, which can be seen as a simple implementation[Fleming, 2008]. The cryptographic algorithm RSA using a residue number system managed to deliver a speedup of 3 compared to the CPU processing time but to achieve this speedup, it needs to perform a large number of parallel exponentiations.[Fleming, 2008]
In the simulation world, the GPPGU is a natural choice as it needs to perform enormous calculations as they process an enormous set of data that can be easily translated into the GPU programming style but still presents the drawback to passing this information back to the CPU to be then presented to the user or the programmer.
As motherboards improve these values, we always are decaying this drawback, and if we make these simulations in the new Apple Chip M1, we eliminate this drawback at.

3.3.7 Programming Languages

Metal

Metal provides a well-optimized platform and low-overhead API to develop three-dimensional applications using a highly integrated language between graphics and computer programs.
This API, as developed by Apple, manages to present better results in terms of

efficiency for the entire Apple ecosystem.

Metal is a unified, low-overhead, low-level API for the graphics processing unit. It is unified because it can be used both for three-dimensional and parallel data processing.

It is a low-level API because its language is almost designed for direct access to the GPU. This API is low-overhead because it reduces runtime through multithreading and pre-compiled resources.

This API presents outstanding reliability in its programming, with feedback at each step, and through great help tools, the API Validation and the Shader Validation.

The API Validation is a tool that checks if the code that calls the Metal API is incorrect, including errors in the creation of resources, in the encoding of commands and other types of commands.

Shader Validation is a tool that detects "out-of-bounds" memory accesses and attempts to access null textures. It is also responsible for identifying all errors in the "command buffer."

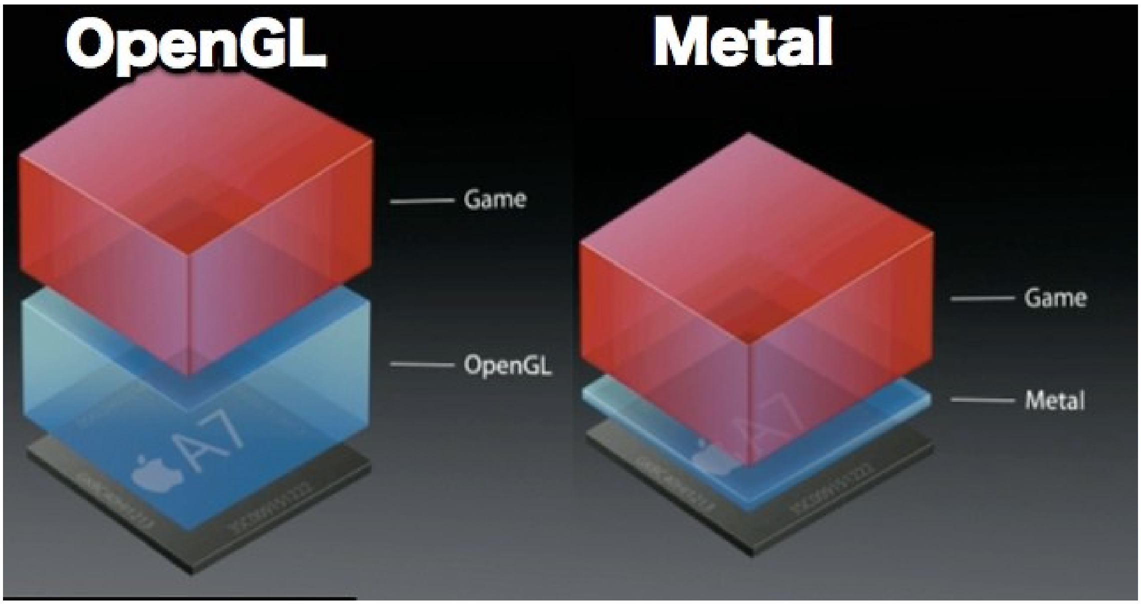


Figure 3.9: Overhead comparison between OpenGL and Metal API
Source: [Anandtech]

Vulkan

Vulkan is an Operating System (OS)-independent programming API, supported on Windows, Linux, and Android, with low overhead and 3D graphics processing.

This API is currently supported on Windows and Linux. However, there is a translation library for macOS, MoltenVK. This open-source library allows Vulkan applications to be executed with some limitations since it translated into the Metal API.

This API is intended to offer better performance, a better balance between CPU and GPU processing, and is capable of parallel processing.

As it is the latest low-level API on the market, with the first stable version released on October 13, 2021, it may have several development flaws.

As it is the latest low-level API on the market, with the first stable version released on October 13, 2021, it may have several development flaws.

Vulkan is not backward compatible with the most common API utilized in GPU, OpenGL, although there is a library, Mesa, that is an implementation of OpenGL/-GLES that runs on top of Vulkan, called Zink.

Vulkan can be described as:

- A Unified API, since it works on both desktop and mobile graphic devices.
- Cross platform due being available on modern operative systems like Android, Linux, BSD Unix, QNX, Nintendo Switch, Raspberry Pi, Stadia, Fuchsia, Tizen, Windows 7, 8, 10, and 11 and with the MoltenVK API it also supports macOS, iOS and tvOS by wrapping over the Metal API.
- Low CPU usage sue the user of batching and other low-level optimizations to reduce the CPU workload.
- Multi-threaded friendly design since usually GPU APIs do not scale well on multicores, Vulkan offers to improve scalability on multicore CPUs due to its modern threading architecture.
- Pre-compiled shaders do not have a long loading screen so that the system can compile the shader code. These APIs drivers are supported to ingest shaders already translated into an intermediate binary format called SPIR-V(Standard Portable Intermediate Representation).

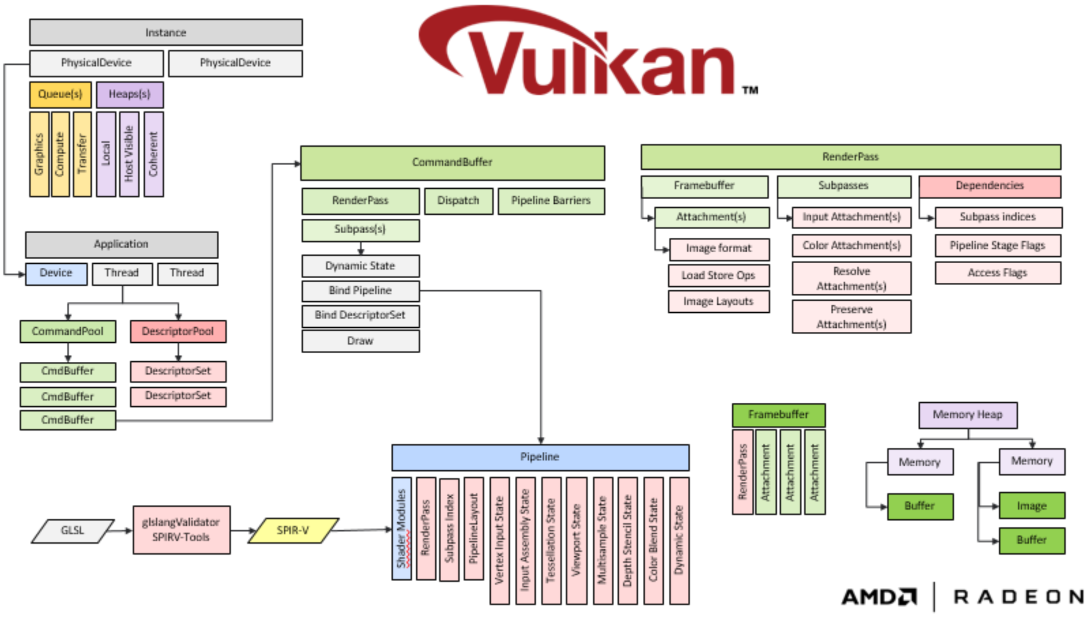


Figure 3.10: Vulkan API
Source: [Gpuopen, 2018]

OpenGL

OpenGL is the most used 2D and 3D programming API in the industry, as it is one of the oldest. This API is OS independent since it was developed for all operating systems.
OpenGL was designed at the initial stage of graphics processing programming, being architected as a state machine. This way, its interface differs from the rest of the APIs.
Although it was released on June 30, 1992, it only released a stable version on July 31, 2017.
OpenGL is no longer in active development since 2006 OpenGL has been managed by the Khronos Group, which owns Vulkan. The main reason that it is not being developed is because of its old architecture that does not support modern GPU capabilities like ray tracing, which is a must in nowadays computer graphics languages.
Being one of the first GPU programming languages have a good advantage, its documentation, it is very well documented, and since its development is stopped, it completed.
OpenGL can be instantiated with C, C++, Python, Java, and more, so software development has various uses.
Figure 3.11 is a diagram of how OpenGL works in the system. The orange rectangles represent the system devices. The light color blue is used to represent the GPU system parts. The green color represents the libraries utilized in the Linux OS. Finally, the objects represented in navy blue are the application components that will utilize the API.

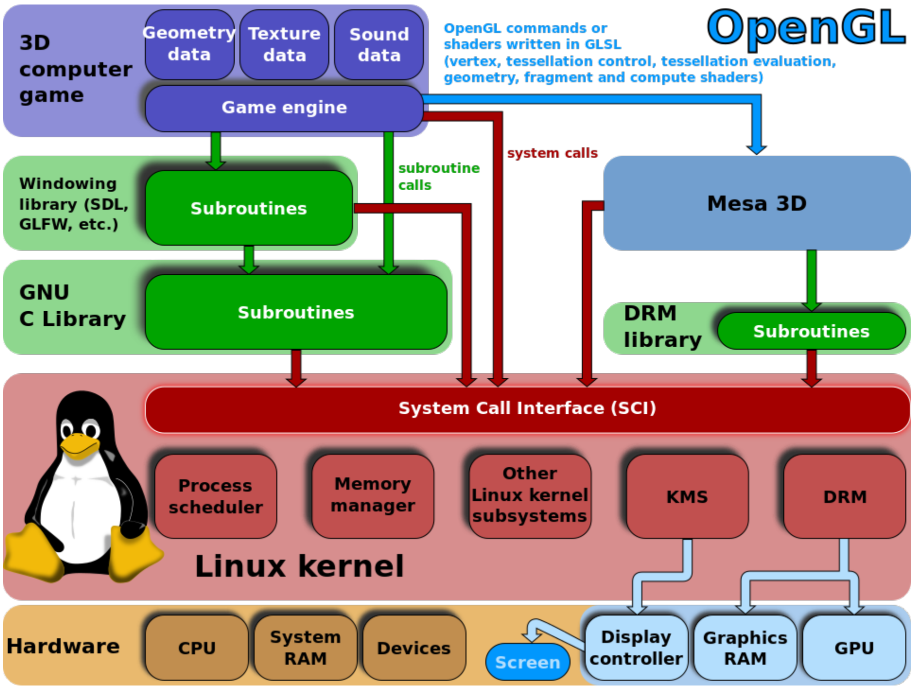
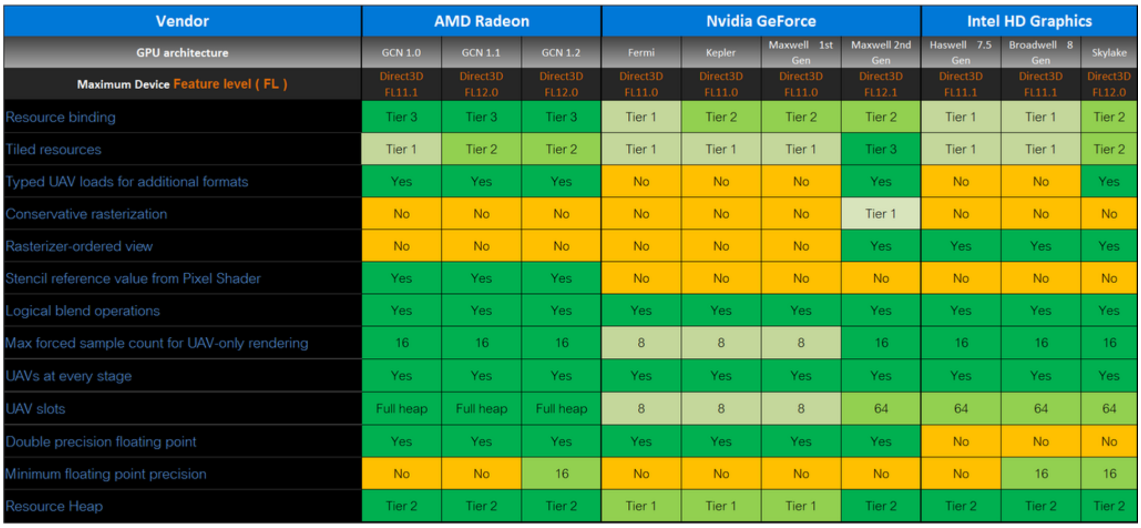


Figure 3.11: OpenGL Architecture
Source: [Wikipedia contributors, 2022c]

DirectX

Microsoft DirectX is a collection of APIs for handling tasks related to multimedia, especially game programming and video, that only can be used in the Windows operative system.
The DirectX contains the Microsoft DirectCompute API, which supports running compute kernels on general-purpose computing on graphics processing units on Microsoft's Windows Vista, Windows 7, and later versions.[Wikipedia contributors, 2019]
The DirectCompute kernel language is very similar to HLSL and is highly identical to C but in a high-level shader language to achieve good performance.
Since DirectX is a collection of APIs for handling tasks related to multimedia. An SDK(Software Development Kit) consists of runtime libraries redistributed in binary form along with its documentation and headers so that it can be an advantage for software development.
Since DirectX, its compatibility is restricted to the Windows OS, but older software versions can still be utilized in modern windows versions.



Vendor	AMD Radeon			Nvidia GeForce				Intel HD Graphics		
GPU architecture	GCN 1.0	GCN 1.1	GCN 1.2	Fermi	Kepler	Maxwell 1st Gen	Maxwell 2nd Gen	Haswell 7.5 Gen	Broadwell 8 Gen	Skylake
Maximum Device Feature level (FL)	Direct3D FL11.1	Direct3D FL12.0	Direct3D FL12.0	Direct3D FL11.0	Direct3D FL11.0	Direct3D FL11.0	Direct3D FL12.1	Direct3D FL11.1	Direct3D FL11.1	Direct3D FL12.0
Resource binding	Tier 3	Tier 3	Tier 3	Tier 1	Tier 2	Tier 2	Tier 2	Tier 1	Tier 1	Tier 2
Tiled resources	Tier 1	Tier 2	Tier 2	Tier 1	Tier 1	Tier 1	Tier 3	Tier 1	Tier 1	Tier 2
Typed UAV loads for additional formats	Yes	Yes	Yes	No	No	No	Yes	No	No	Yes
Conservative rasterization	No	No	No	No	No	No	Tier 1	No	No	No
Rasterizer-ordered view	No	No	No	No	No	No	Yes	Yes	Yes	Yes
Stencil reference value from Pixel Shader	Yes	Yes	Yes	No	No	No	No	No	No	No
Logical blend operations	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Max forced sample count for UAV-only rendering	16	16	16	8	8	8	16	16	16	16
UAVs at every stage	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
UAV slots	Full heap	Full heap	Full heap	8	8	8	64	64	64	64
Double precision floating point	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No
Minimum floating point precision	No	No	16	No	No	No	No	No	16	16
Resource Heap	Tier 2	Tier 2	Tier 2	Tier 1	Tier 1	Tier 1	Tier 2	Tier 2	Tier 2	Tier 2

Figure 3.12: DirectX 12 / Direct3D 12 Feature Levels and Resources
Source: [Overclock, 2015]

CUDA

CUDA or Compute Unified Device Architecture is a parallel computing platform and API that allows software that can be used in GPGPU.
The NVIDIA company developed CUDA, and it was designed to work with programming languages such as C, C++, and Fortran but can also be utilized in Python language for quick development with the Numba compiler.
Since it has, NVIDIA created it, it can only be utilized in their GPU devices, but since there is a wide variety in modern society with computers with their graphics cars, it would still limit a narrower amount of computers. With Apple's new M1 chip, we currently have three primary graphical card providers, NVIDIA, AMD, and Apple, in the graphics cards market.
CUDA was launched in 2006 with over 150 CUDA-based libraries, SDKs, and profiling and optimization tools, but they are constantly innovating, and currently, thousands of GPU-accelerated applications are built on CUDA.
It offers the programmer flexibility and programmability, so it is one of the choices used for researching and deploying new deep learning and parallel computing algorithms.

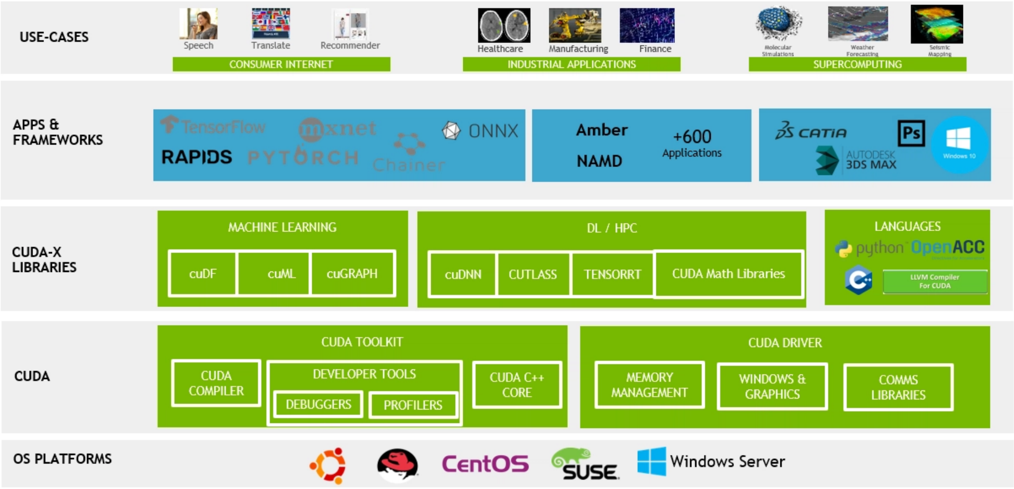


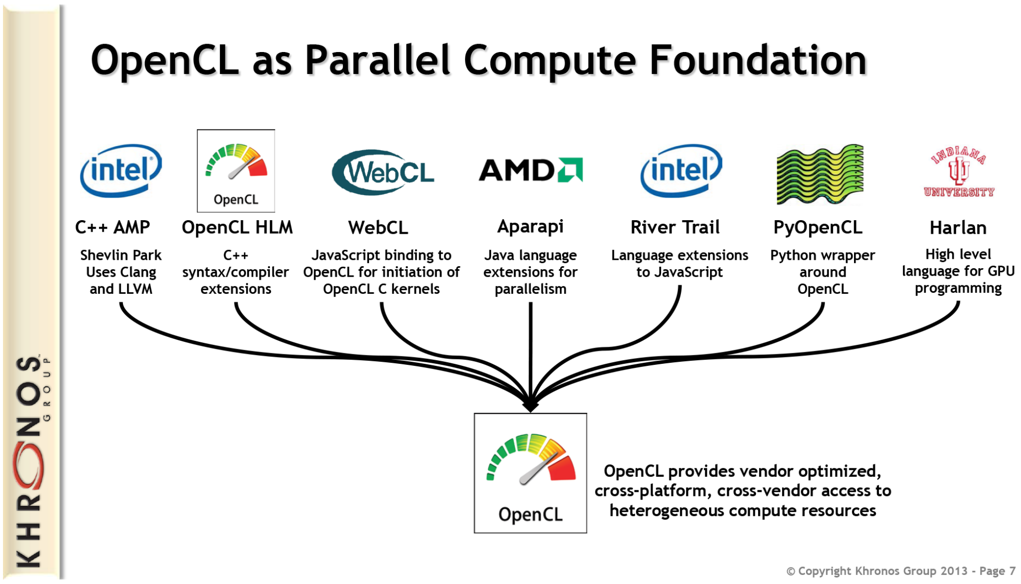
Figure 3.13: CUDA tools, libraries, frameworks, and use-cases
Source: [NVIDIA, 2012]

OpenCL

OpenCL (Open Computing Language) is an open-source framework, parallel programming of various accelerators in supercomputers, cloud servers, and personal computers. It dramatically improves the speed and responsiveness of a broad spectrum of applications in numerous market categories.
It has maintained by the non-profit technology consortium Khronos Group and can be executed in AMD, ARM, Intel, and NVIDIA devices. With the introduction of the Metal API, Apple deprecated OpenCL with macOS 10.14, released on September 24, 2018.
The main difference between OpenGL and OpenCL is that OpenGL enables programming to do graphical operations, and OpenCL allows programming to do the computation in multiple processors.

3.3.8 Comparison of API's and the API chosen

In order to migrate the audio render engine in Sound Particles to the GPU, it will need a API that is up to date with the technological market. Given the large percentage of the company's users that use the macOS OS, the choice of API to adopt must allow this operative system, at least.
At the moment, only OpenGL can run natively on all operating systems (Windows, macOS, and Linux). However, it remains an outdated API in today's world.
Vulkan is a highly recent language, which released its stable version in October 2021 and can be seen as too recent an API for mitigating the enormous power needed to migrate the current Sound Particles engine to the GPU.
Metal, despite being an API closed to the Apple ecosystem, is a low-level language that has great processing efficiency and excellent documentation.



OpenCL as Parallel Compute Foundation

C++ AMP
Shevlin Park Uses Clang and LLVM

OpenCL HLM
C++ syntax/compiler extensions

WebCL
JavaScript binding to OpenCL for initiation of OpenCL C kernels

Aparapi
Java language extensions for parallelism

River Trail
Language extensions to JavaScript

PyOpenCL
Python wrapper around OpenCL

Harlan
High level language for GPU programming

OpenCL provides vendor optimized, cross-platform, cross-vendor access to heterogeneous compute resources

© Copyright Khronos Group 2013 - Page 7

Figure 3.14: CUDA tools, libraries, frameworks, and use-cases
Source: [NVIDIA, 2012]

CUDA will limit the use of the engine by a significant window(only NVIDIA devices), so only users with this kind of graphics card would be able to run this optimization to the engine so that it will be discarded as a viable option.
DirectX is also eliminated for the same reasons as the CUDA, but this time it only can be utilized in the Windows operative system.
In the end, we only have two viable APIs on the table, Vulkan and Metal, being both the most utilized API to develop GPGPU software.
The following table demonstrates the main factors of choice that are crucial in order to choose the more indicated API to be utilized in Sound Particles:

lightgray	API's List				
API Name	OS	Efficiency	Documentation of the api	GPU Debugging	Offline Kernel Compilation
Vulkan	Linux, BSD Unix, Windows, macOS,	This API is ready to be utilized in large-scale applications due to its high modernity and performance render benchmarks. "Experiments 5 and 6 prove that OpenGL ES cannot keep up with Vulkan's performance when power is not a concern." [Lujan et al., 2019]	Excellent documentation with sample code from Khronos-Group	Vulkan offers a wide variety of debugging tools to the developer who has a better knowledge of his mistakes and how to fix them. It provides RenderDoc, which is a frame-capture-based graphics debugger. NVIDIA Nsight allows the developer to build and debug integrated GPU kernels and native CPU code and inspect the GPU and memory state. Arm Perfdoc is a cross-platform Vulkan layer that checks Vulkan applications for recommended API usage on Arm Mali devices. The AMD Radeon™ GPU Profiler is a groundbreaking low-level optimization tool that provides detailed information on Radeon™ GPUs.	Yes

Metal	macOS	Metal is a very optimized API that should be utilized for all applications developed for iOS and OS X. It is one of the most efficient API 's in the market due to its low overhead and its high efficiency with Apple chips. "It has been shown that Metal API gives less time rendering for the datasets used in the implementation than OpenGL ES. " [Abdallah et al., 2015]	Good documentation, simple examples for more complex features, and WWDC's (Apple Worldwide Developers Conference) videos with explanations and live demonstrations	Xcode provides a suite of tools that enable the developer to inspect, debug, and profile the application developed by recording the Metal API calls and generating a GPU frame capture. The frame capture feature is a snapshot of every Metal command and data buffer the GPU used to render each call. Each frame capture delivers the commands, shader code, and GPU performance.	Yes
OpenGL	Linux, BSD Unix, Windows, macOS,	This API got deprecated in June 2018, so it should be only utilized as a last resort due to its portability and completed documentation. It was not designed to modern GPU capabilities.	Completed due deprecation with a wide variety of examples	gDEBugger is a powerful OpenGL and OpenGL ES debugger and profiler, delivering one of the most intuitive OpenGL development toolkits for graphics application developers. It helps the developer save precious debugging time and boost the application's performance. It traces the application activity on top of the OpenGL API to provide the necessary data to find bugs and optimize application rendering performance.	No

DirectX	Windows and Linux	DirectX is the most widely used graphics API due to all major video game providers developing their games in Windows. Since DirectX 12 was released, few articles have described its performance. “This is a simple fact that DirectX 12 is brand-new, and it will take time for developers and graphics vendors to optimize their use of it.” [Aaron Klotz, 2022]	Well documented, offering regularly updated documentation and helpful developer communities, providing a starting guide but confusing to discover critical components.	We need an editor to use the debugging app, and developers can use Visual Studio and the Windows 8 SDK to debug DirectX apps remotely. The Windows 8 SDK includes components that support DirectX development and provide error checking with parameters. Effect-Compiler Tool is an offline executable tool for compiling HLSL shaders for all the respective versions of Direct3D.DxDiag helps us identify problems related to audio, display, video, and any other multimedia applications with required features running on our computer.	Yes
CUDA	Nvidia graphics cards	CUDA is widely used due to its high portability with NVIDIA cards, and since the NVIDIA company developed it, it has delivered the best performance possible for its video cards. “It improves performance by up to 50% on the Intel-Volta/Pascal-PCI-E platform but brings little benefit to the Power9-Volta-NVLink platform.” [Chien et al., 2019]	Good documentation with a wide variety of tips and examples	CUDA-GDB is the primary debugging tool that supports debugging of both 32 and 64-bit CUDA C/C++ applications. It provides complete control over the execution of the CUDA application, including breakpoints and single-stepping. It provides the analysis of variables, read/write memory, and registers and inspects the GPU state when the application is suspended.	Yes

OpenCL	Linux, Windows, macOS	OpenCL is deprecated, so its performance will not increase and is stable, and for this reason, like OpenGL, it should be utilized only as a last resource since its architecture does not support modern GPU capabilities "Experimental results on an AMD/ATI HD5850 GPU for a set of commonly-used benchmarks show that we achieve 2.1X 6.7X speedup concerning the un-optimized versions ..." [Zhu et al., 2012]	Poor Documentation and not easy for beginners	gDEBugger CL is a debugging tool equivalent to the OpenGL tool but for OpenCL. This tool enables the developer to locate parallel computing performance bottlenecks, edit and continue OpenCL kernels "on the fly," and break on OpenCL errors, function calls, and memory leaks.	No

Metal will be the API of choice, as it benefits the company users so much since with is highly efficient compared to the rest and has an extensive portfolio of documentation. A large percentage of the Sound Particles software utilizes the macOS operative system, and with Apple's new M1 chip it eliminates all the cost of having to transfer the data from the CPU to the GPU to be processed and then be transferred back to the CPU to be utilized in the software.

3.3.9 Disadvantages of explicit API's

- Hard to understand since it requires a deep knowledge of GPU;
- Hard to use since the application is a driver ;
- Hard to be portable, due the API limitations;
- Hard not to explode since incorrect usage leads to Undefined Behaviour (UB).

3.3.10 GPU new concepts

Pipeline

A computer graphics pipeline is a conceptual model that describes what steps a graphics system needs to perform to render the commands encoded by the CPU. However, graphics application programming interfaces (APIs) such as Direct3D and OpenGL were developed with the intuition to unify similar steps and to control the graphics pipeline of a given hardware accelerator.
These APIs abstract the underlying hardware and keep the programmer away from writing code to manipulate the graphics hardware accelerators.
Frequently, most of the pipeline steps are implemented in hardware, which allows for special optimizations. The term "pipeline" stands in a similar sense to the pipeline in processors: the individual steps of the pipeline run in parallel as long as any given job has what it needs.

Command Buffers

Command buffers are the primary mechanism for sending commands from the CPU to be executed on the GPU. Command buffers consist of a container that stores encoded commands for the GPU to perform.
In a multithreaded app, it is advisable to break jobs that need to be accomplished into subtasks that can be encoded separately.
So we need to create a command buffer for each render job. Then, if we need to establish the execution order, we need to synchronize the command buffers' en queue since the pipeline queue follows a FIFO methodology.
The command queue automatically schedules and executes these command buffers as they become available.
By following the best practices described below, we can achieve performance gains on both the CPU and the GPU by maximizing parallelism, avoiding bottlenecks, and reducing idle times on the GPU.

- Embrace the responsibility for achieving and controlling GPU/CPU parallelism, so do not expect that the API performs magic in achieving the best parallelism possible;
- Develop a multithreaded application to split the workload between all CPU threads;
- Initialize, the command queue and all buffers required to the application at startup so that the application does not suffer from the cost of creating them;
- In the main GPGPU API 's like Metal, Vulkan, and CUDA, we can overlap graphics or compute work;
- Compute and graphics workloads can be scheduled together, and we can use fences (Semaphores in GPGPU) to synchronize the workloads;

- Do not wait until the end of the frame to submit the command buffer to the GPU. It might result in stopping the parallelism between the CPU and GPU, instead create an asynchronous task to utilize the command buffer result after it has concluded;
- Do not create too many CPU threads, since it can lead to insolvency in the CPU resources.

3.3.11 Metal API

In Figure 3.15, we can observe the Metal objects hierarchy since all of its objects depend on the object above in the hierarchy.

The following sections are a brief description of the most important objects that will be utilized during the render engine migration.

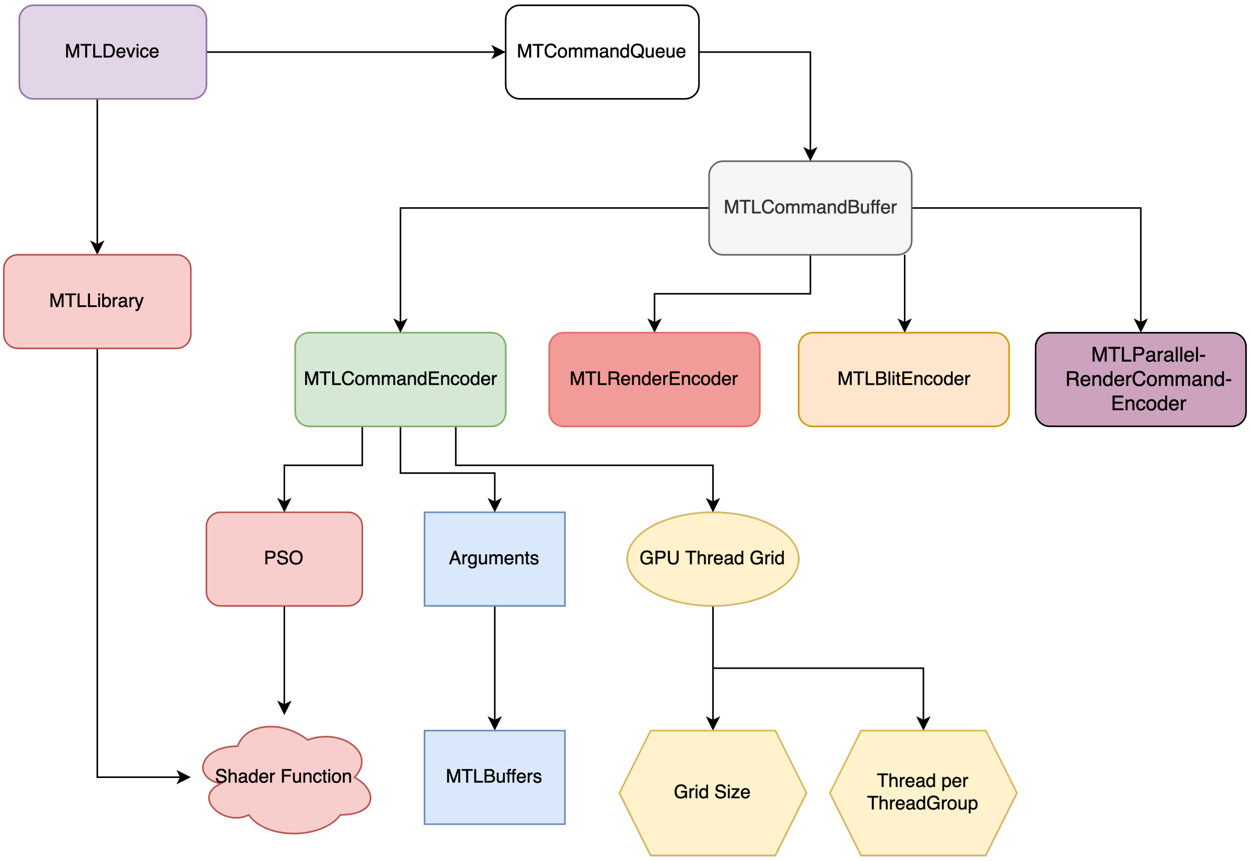


Figure 3.15: Metal Objects Hierarchy
Source: Author

MTLDevice

This is an abstraction of the physical GPU that will consume the rendering and compute commands. This is the go-to object to do anything in the Metal API since all objects the application interacts with come from this object.

MTLFunction

It is an object that represents a public shader function in a Metal library. There are two kinds of shaders in the MSL. In compute operations, we utilize a compute function (also known as a compute kernel), which performs a parallel calculation using a grid of threads. If the task is to render graphics, we will utilize the vertex and fragment shaders.

MTLLibrary

The MTLLibrary is an object that contains all the Metal shaders compiled into a single library. We will use this to fetch our required shader functions to define a pipeline state.

MTLBuffer

A resource that stores data in a format that the application defines can be used only with the MTLDevice that created it. It can contain the following scalar data types: bool,int8_t,uint8_t,int16_t,uint16_t,int32_t,uint32_t,int64_t,uint64_t. It can also contain Vector Data Types: bool, char, short, int, long, uchar, ushort, uint, ulong, half, and float. This resource can have four different types of storage mode options:

- **MTLStorageModeShared**: The resource is stored in system memory and is accessible to both the CPU and the GPU.
- **MTLStorageModePrivate**: The resource is in the GPU and can only be accessed by it.
- **MTLStorageModeManaged**: The CPU and GPU may maintain separate copies of the resource, and any changes must be explicitly synchronized.
- **MTLStorageModeMemoryless**: The resource's contents can be accessed only by the GPU and only exist temporarily during a render pass.

MTLCommandQueue

This object stores all the commands and allows the application to control the execution of all commands in a first-come-first-serve(FIFO) order. This also includes the operative system command buffers to render graphics for the display.

MTLCommandBuffer

This object stores translated hardware commands ready for consumption by the GPU. This container contains data for GPU computation. Data is encoded (so that the GPU can read them) and added to Command Buffers by Command Encoders. Buffers can be enqueued (append) or committed (append and execute) to the Command Queue.

MTLCommandEncoder

This object is responsible for translating rendering and compute commands into hardware commands. While a command encoder is active, it has the exclusive right to append commands to its command buffer. Once it finishes encoding commands, call the endEncoding method to complete the command encoder if it is necessary to create a new command encoder to write further commands into the same command buffer.

MTLRenderEncoder

The object for encoding commands for a render pass is primarily used in graphics rendering.

MTLComputeCommandEncoder

An object for encoding commands in a compute pass is utilized for all computations needed for the application.

MTLBlitEncoder

It is an encoder for memory copying, filtering, and filling commands. It is the main encode utilized in all memory management operations in the system, especially with private GPU buffers, since the CPU can not access these buffers created in the GPU.

MTLParallelRenderCommandEncoder

This object consists of a multiheaded operation for a batch of MTLRenderEncoder, that splits up a single render pass so that it can be simultaneously encoded from multiple threads

GPU Thread Grid

To launch the GPU threads, the CPU needs to encode how many threads it will be launched with the method "dispatchThreadgroups:threadsPerThreadgroup:". This command will launch with two variables, both a three dimensions integer containing the three dimensions "threadgroupsPerGrid", and Metal will try to fit as many threads as possible so that it will not need to launch unused threads. The figure 3.16 we can observe the new threads, so we need, if possible, to launch a multiple of the "maxThreadsperThreadGroup" device variable and try not to go above or below it.

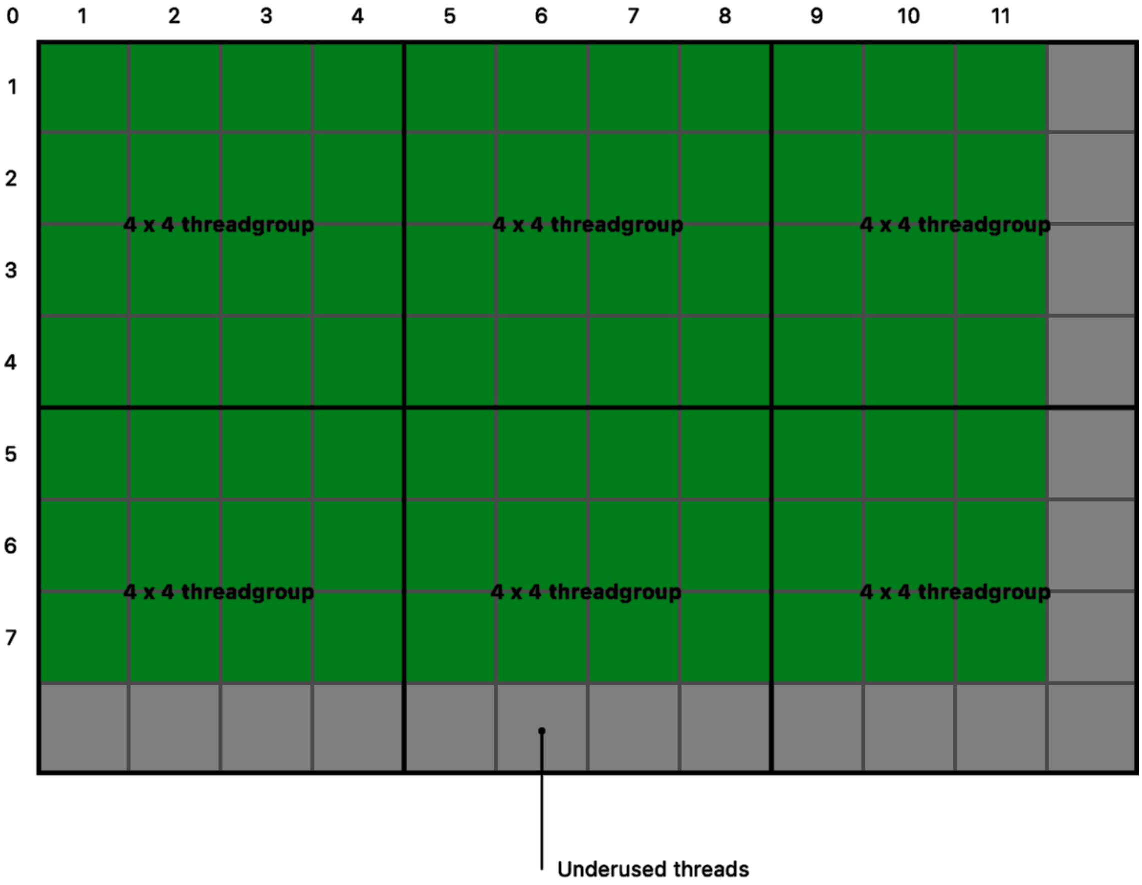


Figure 3.16: Metal grid of threads
Source: [Apple, a]

Chapter 4

System Design

4.1 Requirements

The following requirement is some user stories made by the Sound Particles company and adapted since they were a draft. Furthermore, I added the missing user stories required to the system:

1. **User Storie 21.1 GPU LIST:** As a dev, I want a class that manages the available GPUs and their information.

2. **User Storie 21.2 RANDOM DATA:** As a dev, I want to have random data available in the GPU

3. **User Storie 21.3 AUDIO DATA:** As a software programmer, I want to have all audio stream data available by the GPU.

4. **User Storie 21.4 SYSTEM DATA:** As a software programmer, I want to have all groups and microphone system information in the GPU.

5. **User Storie 21.5 RENDER DATA:** As a software programmer, I want all render data to be available on the GPU.

6. **User Storie 21.6 RENDER METAL:** As a software programmer, I want to create a path to render audio with Metal.

7. **User Storie 21.7 RENDER METAL:** As a software programmer, I want to update the render settings with user preferences like buffer size, frame rate, sound velocity, etc.

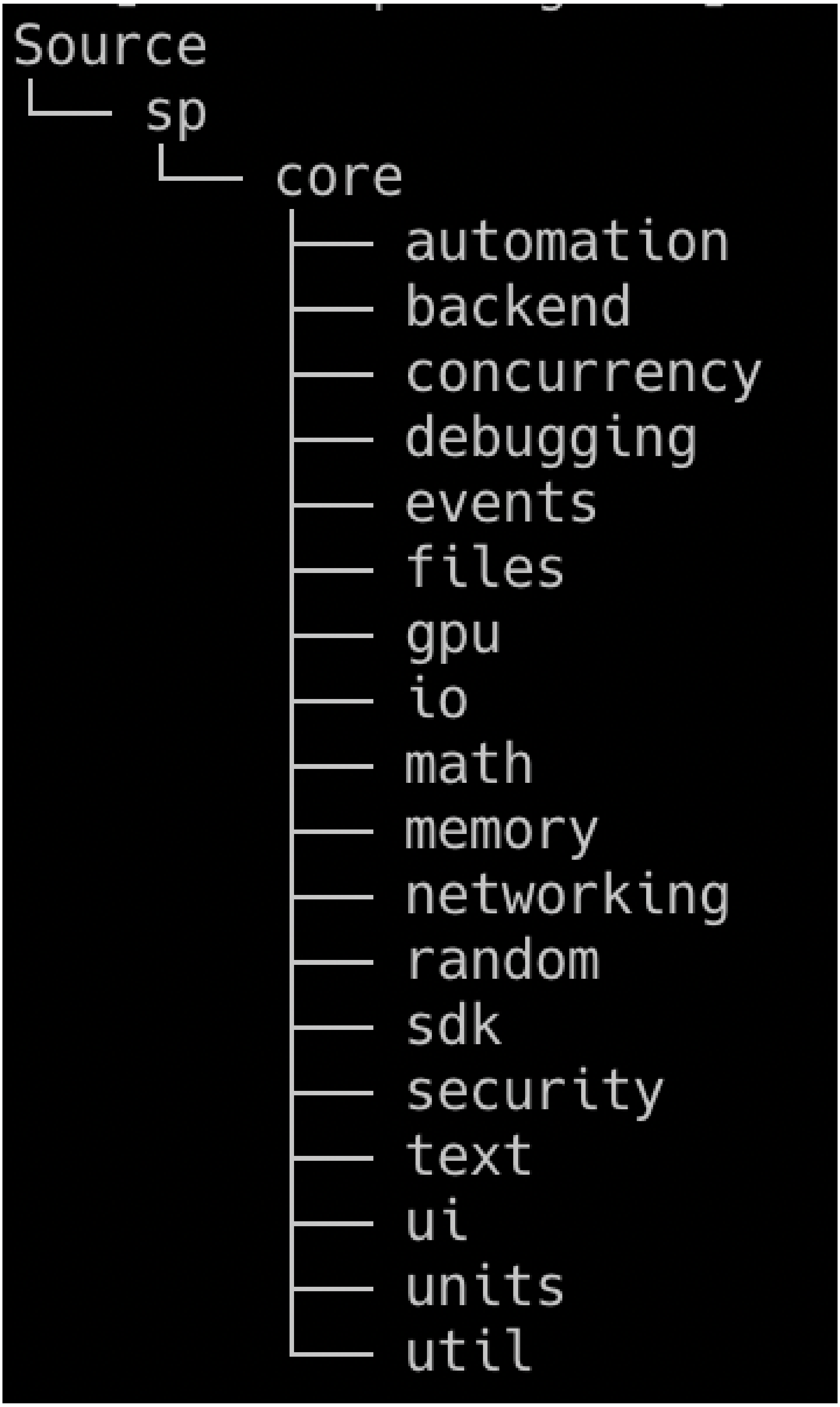
8. **User Storie 21.8 RENDER METAL:** As a software programmer, I want to test the GPU audio result.

9. **User Storie 21.9 RENDER METAL:** As a software programmer, I want to test the GPU audio result.

4.2 Architecture

When I started migrating the render engine to the GPU, a small but fundamental part was already developed.
The bridge from Sound Particles and the Metal API was already developed. The particle-to-microphone polar response was developed for all the microphones, except for the ambisonics type microphone type, although this development was in Objective-C++, a mix between Objective C and C++. This means that part of the code was developed in Objective C, like the direct calls to the GPU, but data types and flow were developed. The best way to describe it is a C++ code that all direct calls to Metal were made in Objective-C because it was the only option since they did not utilize Swift.
The first months, the main focus was to be comfortable around the sp-engine, render engine, and sp-core.After the first month, I started to migrate the current work to C++ since Apple released a version of pre-compiled Metal headers in C++. This provided an implementation more in sequence with the rest of the code since it is written in C++, providing better workflow.
After this migration, I started to migrate the rest of the render operations like computation of particle position, microphone position, sound attenuation, and interpolation. Since so far, the only part being processed in the GPU was the polar response, throughout the migration, minor adjustments to the current architecture were required but not groundbreaking since the initial goal was to follow the work that was already developed.
When the render engine migration was completed, the results were insufficient, so the architecture needed a complete refactoring process to achieve the impressive results that the GPU is known for, referred to in the State of Art of this work. To achieve these results and still utilize the GPU in the live render, the render engine throughput needed to stop processing a particle for a specific microphone at a time and increase the payload to the maximum possible particles at a time for given frame size.
The Figure includes the command "addCompletedHandler" this command Apple describes its function as "Registers a block of code that Metal calls immediately after the GPU finishes executing the commands in the command buffer." [Apple, b]. It was utilized mainly after the interpolation task to copy the result buffer from the GPU to the CPU buffer. It creates a temporary thread to execute the block of code inside, which is released after completion.
The Figure 4.1 is a tree image of the sp-core for a better understanding of all its operations. The figures 4.2, 4.3 and 4.4 are an illustration of the sp-engine tree representing its functionalities and importance to the Sound Particles software.

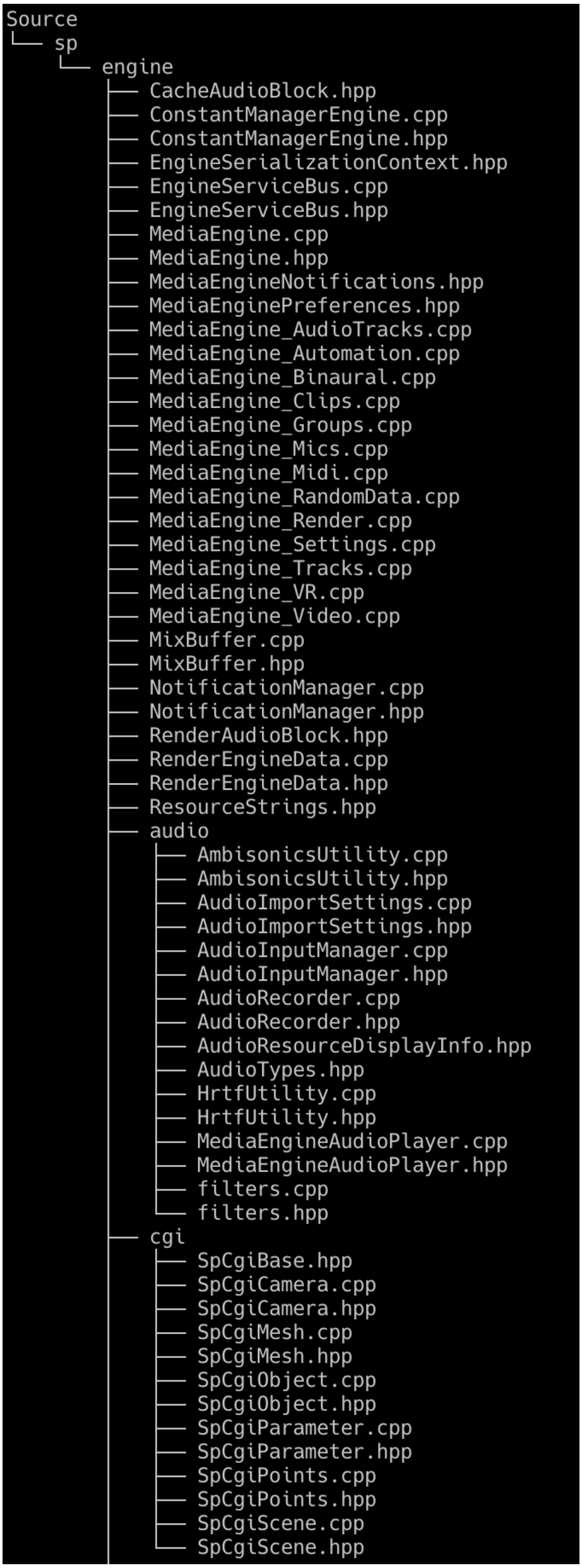
The image 4.5 shows the Sound Particles architecture indicating its modules and the Juce module, an open-source, cross-platform C++ application framework.
JUCE is the one who makes the bridge between the generic Graphical User Interface (GUI) of C++ for the native GUI of each operating system. It is also used to accomplish the abstraction of each audio driver (from CoreAudio + Windows Drivers), and also, at this moment, it is the one that deals with the Sound Particles filesystem.
To summarize, Sound Particles utilizes JUCE as an API to abstract everything na-



```
Source
└── sp
    └── core
        ├── automation
        ├── backend
        ├── concurrency
        ├── debugging
        ├── events
        ├── files
        ├── gpu
        ├── io
        ├── math
        ├── memory
        ├── networking
        ├── random
        ├── sdk
        ├── security
        ├── text
        ├── ui
        ├── units
        └── util
```

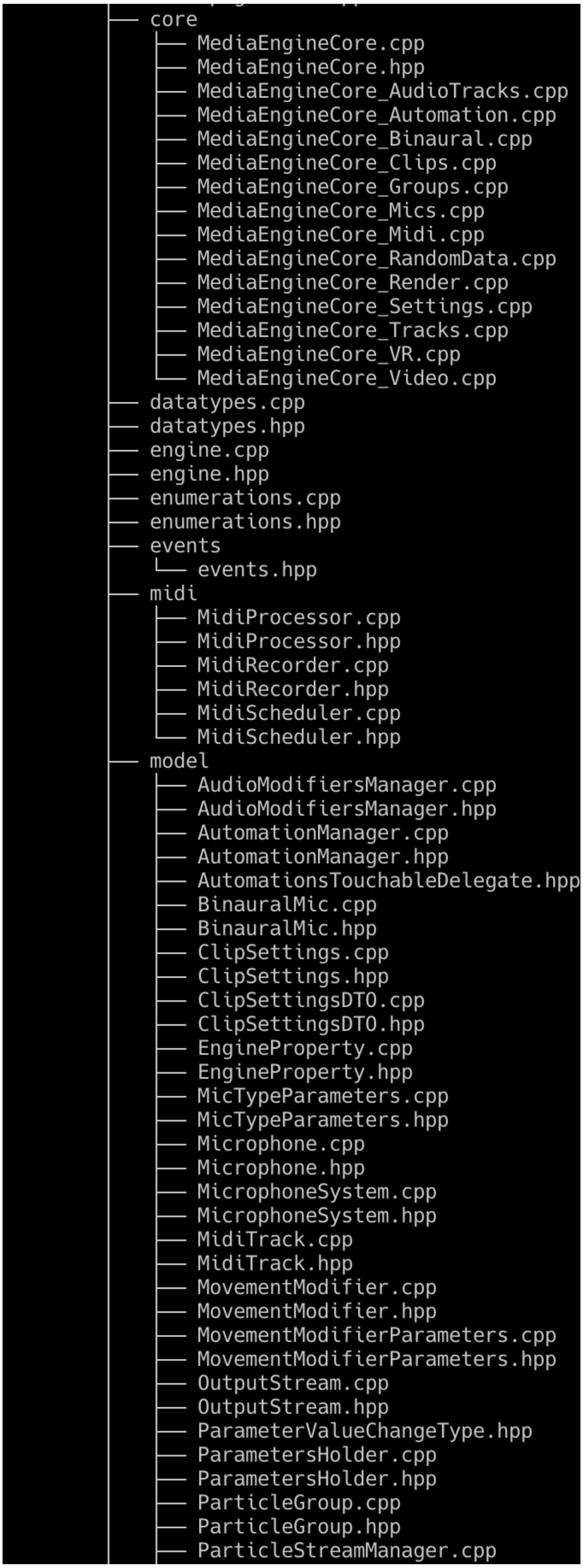
Figure 4.1: Module tree of the sp-core

tive between Windows and macOS.
In the image 4.1, we can observe the importance of the sp-core since it contains all functions required by the application and is represented as one of the most prominent objects in the diagram. We can observe as well that all the audio modules are connected to the sp-engine because the computing process of the resulting sound will need the audio samples provided by the user, so it is one of the essential modules in the system.



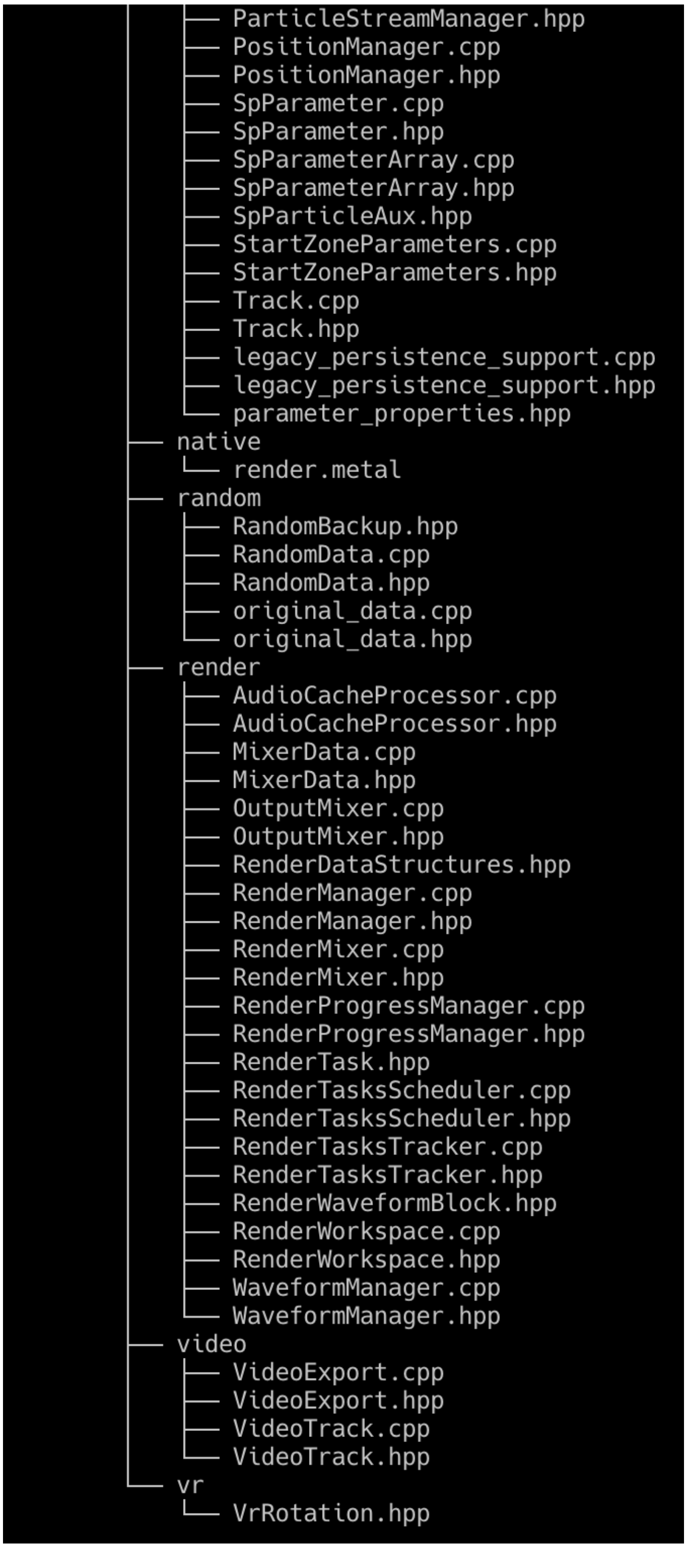
```
Source
└── sp
    └── engine
        ├── CacheAudioBlock.hpp
        ├── ConstantManagerEngine.cpp
        ├── ConstantManagerEngine.hpp
        ├── EngineSerializationContext.hpp
        ├── EngineServiceBus.cpp
        ├── EngineServiceBus.hpp
        ├── MediaEngine.cpp
        ├── MediaEngine.hpp
        ├── MediaEngineNotifications.hpp
        ├── MediaEnginePreferences.hpp
        ├── MediaEngine_AudioTracks.cpp
        ├── MediaEngine_Automation.cpp
        ├── MediaEngine_Binaural.cpp
        ├── MediaEngine_Clips.cpp
        ├── MediaEngine_Groups.cpp
        ├── MediaEngine_Mics.cpp
        ├── MediaEngine_Midi.cpp
        ├── MediaEngine_RandomData.cpp
        ├── MediaEngine_Render.cpp
        ├── MediaEngine_Settings.cpp
        ├── MediaEngine_Tracks.cpp
        ├── MediaEngine_VR.cpp
        ├── MediaEngine_Video.cpp
        ├── MixBuffer.cpp
        ├── MixBuffer.hpp
        ├── NotificationManager.cpp
        ├── NotificationManager.hpp
        ├── RenderAudioBlock.hpp
        ├── RenderEngineData.cpp
        ├── RenderEngineData.hpp
        ├── ResourceStrings.hpp
        ├── audio
        │   ├── AmbisonicsUtility.cpp
        │   ├── AmbisonicsUtility.hpp
        │   ├── AudioImportSettings.cpp
        │   ├── AudioImportSettings.hpp
        │   ├── AudioInputManager.cpp
        │   ├── AudioInputManager.hpp
        │   ├── AudioRecorder.cpp
        │   ├── AudioRecorder.hpp
        │   ├── AudioResourceDisplayInfo.hpp
        │   ├── AudioTypes.hpp
        │   ├── HrtfUtility.cpp
        │   ├── HrtfUtility.hpp
        │   ├── MediaEngineAudioPlayer.cpp
        │   ├── MediaEngineAudioPlayer.hpp
        │   ├── filters.cpp
        │   └── filters.hpp
        ├── cgi
        │   ├── SpCgiBase.hpp
        │   ├── SpCgiCamera.cpp
        │   ├── SpCgiCamera.hpp
        │   ├── SpCgiMesh.cpp
        │   ├── SpCgiMesh.hpp
        │   ├── SpCgiObject.cpp
        │   ├── SpCgiObject.hpp
        │   ├── SpCgiParameter.cpp
        │   ├── SpCgiParameter.hpp
        │   ├── SpCgiPoints.cpp
        │   ├── SpCgiPoints.hpp
        │   ├── SpCgiScene.cpp
        │   └── SpCgiScene.hpp
        │
```

Figure 4.2: Module tree of the sp-engine - Part 1



```
├── core
│   ├── MediaEngineCore.cpp
│   ├── MediaEngineCore.hpp
│   ├── MediaEngineCore_AudioTracks.cpp
│   ├── MediaEngineCore_Automation.cpp
│   ├── MediaEngineCore_Binaural.cpp
│   ├── MediaEngineCore_Clips.cpp
│   ├── MediaEngineCore_Groups.cpp
│   ├── MediaEngineCore_Mics.cpp
│   ├── MediaEngineCore_Midi.cpp
│   ├── MediaEngineCore_RandomData.cpp
│   ├── MediaEngineCore_Render.cpp
│   ├── MediaEngineCore_Settings.cpp
│   ├── MediaEngineCore_Tracks.cpp
│   ├── MediaEngineCore_VR.cpp
│   └── MediaEngineCore_Video.cpp
├── datatypes.cpp
├── datatypes.hpp
├── engine.cpp
├── engine.hpp
├── enumerations.cpp
├── enumerations.hpp
├── events
│   └── events.hpp
├── midi
│   ├── MidiProcessor.cpp
│   ├── MidiProcessor.hpp
│   ├── MidiRecorder.cpp
│   ├── MidiRecorder.hpp
│   ├── MidiScheduler.cpp
│   └── MidiScheduler.hpp
├── model
│   ├── AudioModifiersManager.cpp
│   ├── AudioModifiersManager.hpp
│   ├── AutomationManager.cpp
│   ├── AutomationManager.hpp
│   ├── AutomationsTouchableDelegate.hpp
│   ├── BinauralMic.cpp
│   ├── BinauralMic.hpp
│   ├── ClipSettings.cpp
│   ├── ClipSettings.hpp
│   ├── ClipSettingsDTO.cpp
│   ├── ClipSettingsDTO.hpp
│   ├── EngineProperty.cpp
│   ├── EngineProperty.hpp
│   ├── MicTypeParameters.cpp
│   ├── MicTypeParameters.hpp
│   ├── Microphone.cpp
│   ├── Microphone.hpp
│   ├── MicrophoneSystem.cpp
│   ├── MicrophoneSystem.hpp
│   ├── MidiTrack.cpp
│   ├── MidiTrack.hpp
│   ├── MovementModifier.cpp
│   ├── MovementModifier.hpp
│   ├── MovementModifierParameters.cpp
│   ├── MovementModifierParameters.hpp
│   ├── OutputStream.cpp
│   ├── OutputStream.hpp
│   ├── ParameterValueChangeType.hpp
│   ├── ParametersHolder.cpp
│   ├── ParametersHolder.hpp
│   ├── ParticleGroup.cpp
│   ├── ParticleGroup.hpp
│   ├── ParticleStreamManager.cpp
```

Figure 4.3: Module tree of the sp-engine - Part 2



```
│   ├── ParticleStreamManager.hpp
│   ├── PositionManager.cpp
│   ├── PositionManager.hpp
│   ├── SpParameter.cpp
│   ├── SpParameter.hpp
│   ├── SpParameterArray.cpp
│   ├── SpParameterArray.hpp
│   ├── SpParticleAux.hpp
│   ├── StartZoneParameters.cpp
│   ├── StartZoneParameters.hpp
│   ├── Track.cpp
│   ├── Track.hpp
│   ├── legacy_persistence_support.cpp
│   ├── legacy_persistence_support.hpp
│   └── parameter_properties.hpp
├── native
│   └── render.metal
├── random
│   ├── RandomBackup.hpp
│   ├── RandomData.cpp
│   ├── RandomData.hpp
│   ├── original_data.cpp
│   └── original_data.hpp
├── render
│   ├── AudioCacheProcessor.cpp
│   ├── AudioCacheProcessor.hpp
│   ├── MixerData.cpp
│   ├── MixerData.hpp
│   ├── OutputMixer.cpp
│   ├── OutputMixer.hpp
│   ├── RenderDataStructures.hpp
│   ├── RenderManager.cpp
│   ├── RenderManager.hpp
│   ├── RenderMixer.cpp
│   ├── RenderMixer.hpp
│   ├── RenderProgressManager.cpp
│   ├── RenderProgressManager.hpp
│   ├── RenderTask.hpp
│   ├── RenderTasksScheduler.cpp
│   ├── RenderTasksScheduler.hpp
│   ├── RenderTasksTracker.cpp
│   ├── RenderTasksTracker.hpp
│   ├── RenderWaveformBlock.hpp
│   ├── RenderWorkspace.cpp
│   ├── RenderWorkspace.hpp
│   ├── WaveformManager.cpp
│   └── WaveformManager.hpp
├── video
│   ├── VideoExport.cpp
│   ├── VideoExport.hpp
│   ├── VideoTrack.cpp
│   └── VideoTrack.hpp
└── vr
    └── VrRotation.hpp
```

Figure 4.4: Module tree of the sp-engine - Part 3

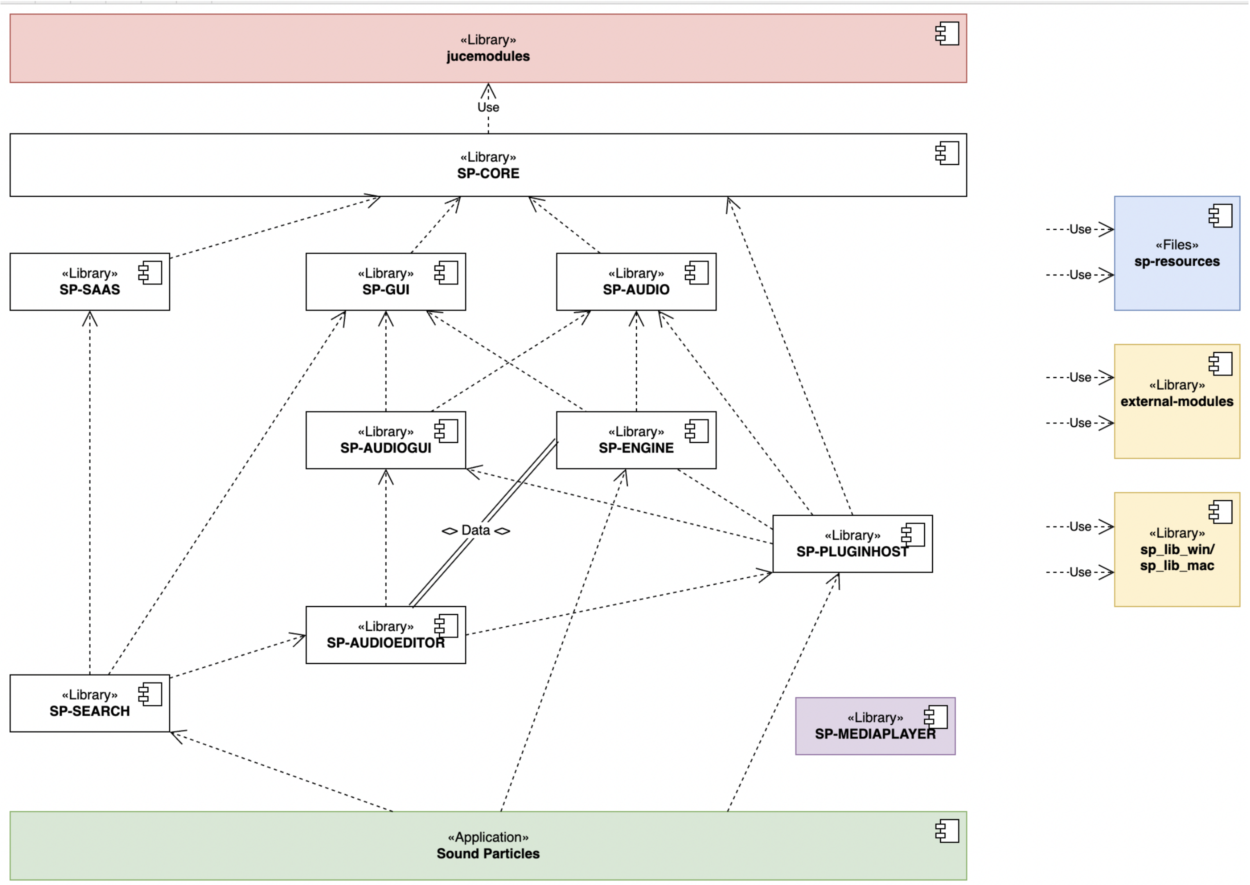


Figure 4.5: Sound Particles Modules

Chapter 5

Implementation

The starting weeks of development mainly focused on studying the render engine and core, all the data types used, and all operations made by the engine to reach the audio result.
The Sound Particles render engine is multithread where it has a pool of render threads, one per CPU core, that will process a render job. This render job is for all microphones in a microphone system to process the particles in a particle group, but it will only process a batch of particles per job. This means that each render job will only process particles from 64 to 64, which means that if its first particle to process is particle 0, the next one is 64, followed by 128 until it reaches the end of the particle number in the group.
In the Sound Particles software, two crucial modules will get refactored, the sp-engine and the sp-core. The sp-engine contains the render engine. The sp-core is the module that combines all the common functionality among all types of applications: serialization, file IO, networking, security, concurrency, and generic algorithms/structures in the parts with more mathematical computation. The sp-core contains a CPU thread that will wait for the requests from the sp-engine and will encode and commit the GPU, and when the render is concluded, this thread is released.
After this study, the development started step-by-step. The following list shows the order of the task development:

1. Create necessary buffers required by the GPU and all the CPU datatypes like CPU buffers in the application startup and all the other queues and devices.

2. Migration of the current objective C++ bridge to the Apple C++ bridge.

3. Compute particle position according to its group type (Circle, Square, Rectangle, Taurus, ...)

4. Compute microphone position

5. Compute microphone direction

6. Finalize the polar response since the ambisonics microphone type was not being executed in the GPU

7. Calculate particle position for each frame if movement exists in the group
8. Compute sound attenuation
9. Compute sound interpolation
10. Compute pitch processing in the audio
11. Refactor of the render architecture to achieve better results

It is also notorious but not mentioned in the list that there was a constant refactoring to the CPU code to integrate the new tasks added to the render path.

The migration until the polar response for the Ambisonics microphone was very straightforward since it was a complete match between the CPU and GPU programming approach.

The Ambisonics polar response needed refactoring due to GPU architecture restrictions like eliminating the while loop and reducing the number of ifs conditions.

One of the final parts of the render system is partitioning the sound frames into audio blocks and processing the interpolation for each one.

For a more efficient system, wider frame size was needed during the interpolation process since the block frame size was concise, so a full frame size was sent to the GPU, and after its completion, the frames were divided into the blocks, which after needed to be sent to the audio mixer process.

The kernels written got also got rewritten several times so that they could achieve better results, like using packed_floats3 instead of the typical float3 for positions. Hence, it simultaneously diminishes memory allocations and increases efficiency slightly.

This first implementation is simply a part-by-part migration of the operations that the CPU render engine needs to execute to achieve the resulting sound. So for each step, a GPU request was made, totaling five requests per render job (particle position, microphone position, and direction, if required a pitch processing, polar response for type of microphone, and a final one for attenuation and interpolation).

This first GPU render implementation did not deliver the desired outcome, a 30 % of the CPU render time, and only resulting in 82%. This difference can be explained by the render engine job architecture restrictions like the render mixer expected a specific frame size to process its job, and it will only process a specific amount of particles, the mixer process did not get migrated to the GPU, and the GPU calls were high with short kernels.

To achieve the desired outcome, a new architecture needed to be implemented from scratch that still follows the CPU engine limitations described above, encapsulating the maximum number of kernels possible to diminish the overhead.

The implementation of this second architecture resulted in the biggest refactor to the engine developed since it completely changed the rendering course.

Firstly, if there exists a pitch modification in the audio, encode a request to the GPU to process it and change the audio buffer accordingly.

Secondly, encodes a request that processes a batch of particles in a particle group, then process all microphones in a microphone system its position and direction, and then compute the polar response and the attenuation.

After it encodes a request that will process the sound interpolation and after it is submitted to the GPU, the CPU will create the audio blocks so that after its completion, it only need to separate the audio sample per block and send it to the audio mixer thread.

This implementation resulted in only two kernels to render the audio in the GPU one that calculates a batch of particles in the particle group and for all microphones process the sound until the interpolation, the second kernel will only process the last part of the interpolation. The only reason it could not be a single kernel to render is that the final part of the interpolation can utilize previous or forward values, and since each thread will process its frame parallelly, it can conflict with other threads. The CPU implementation does not have this issue since it processes the frames sequentially, and if it performs or utilizes a previous value, it is already calculated.

To better understand the architecture developed for the GPU integration in the Sound Particles software, Figure 5.1 is a diagram of how the data is passed between modules and the GPU. In the Figure, the round rectangles represent the system devices, the CPU and the GPU. The straight rectangles represent the system modules, the sp-engine, and the sp-core. The arrows represent the direction of how the data is passed between the modules and system devices.

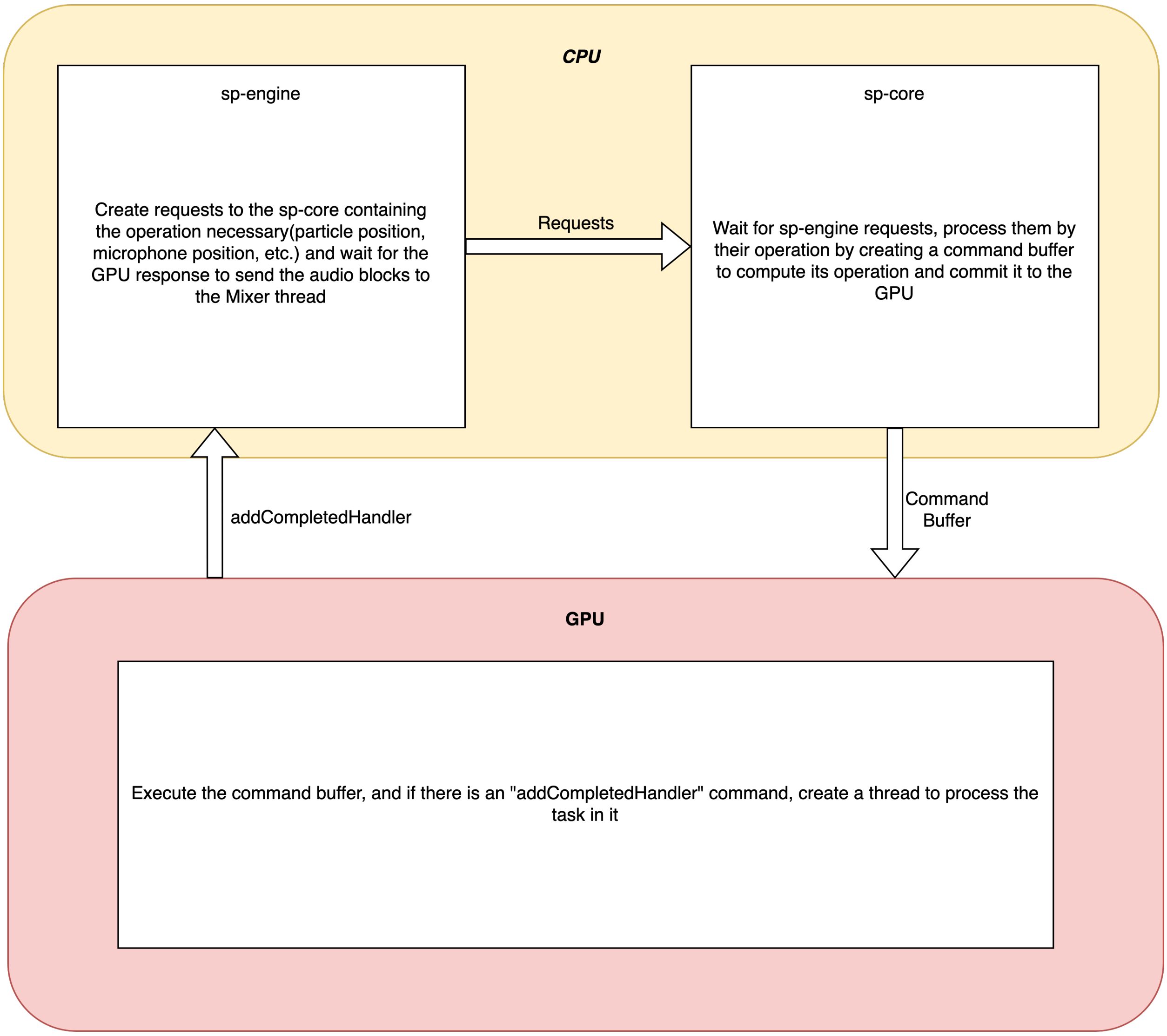

CPU
sp-engine
Create requests to the sp-core containing the operation necessary(particle position, microphone position, etc.) and wait for the GPU response to send the audio blocks to the Mixer thread
Requests
sp-core
Wait for sp-engine requests, process them by their operation by creating a command buffer to compute its operation and commit it to the GPU
addCompletedHandler
Command Buffer
GPU
Execute the command buffer, and if there is an "addCompletedHandler" command, create a thread to process the task in it

Figure 5.1: Diagram of the gpu architecture integration in the Sound Particles software

Source: Author

Chapter 6

Results, analysis, and tests

6.1 Results and analysis

6.1.1 First Semester

In the first semester, the work developed was the prototypes, and to obtain only the render results, the benchmark testing started only when the render process started until it ended, eliminating all variables initialization in the system.
The figure 6.1 demonstrates that the results achieved were auspicious since it processed in a single shader all the particles in the system(all particle groups at once). Since I was instructed to change it because it will not scale well in a large number of particles in the system, this phase was concluded with a revision of the architecture developed to achieve a better scaling architecture.

6.1.2 Second Semester

In the second semester, one of the first steps, after I started the development, was to develop two benchmark tests for the CPU render time and another for the gpu. Figure 6.2 represents the baseline tests made: Four types of microphones systems(stereo, ambisonics, a microphone system with six microphones(5.1), and a microphone system with eight microphones(7.1); With a particle emitter that emits ten particles per second for a duration of one hundred seconds that will make a sound that lasts five seconds, a particle group with one thousand particles that will radiate a sound that lasts five seconds and finally a particle group with ten thousand particles that will cast a sound that lasts one second; And presents the respective initial results for each render engine as well.
The figures 6.3 and 6.4 present all the versions developed during the internship and display the render duration at each step in seconds.
The figure 6.5 represents a graphical image of how the GPU render time changed each version of the render engine migration.
The sixth version was just a test to verify the added time if the sp-core thread

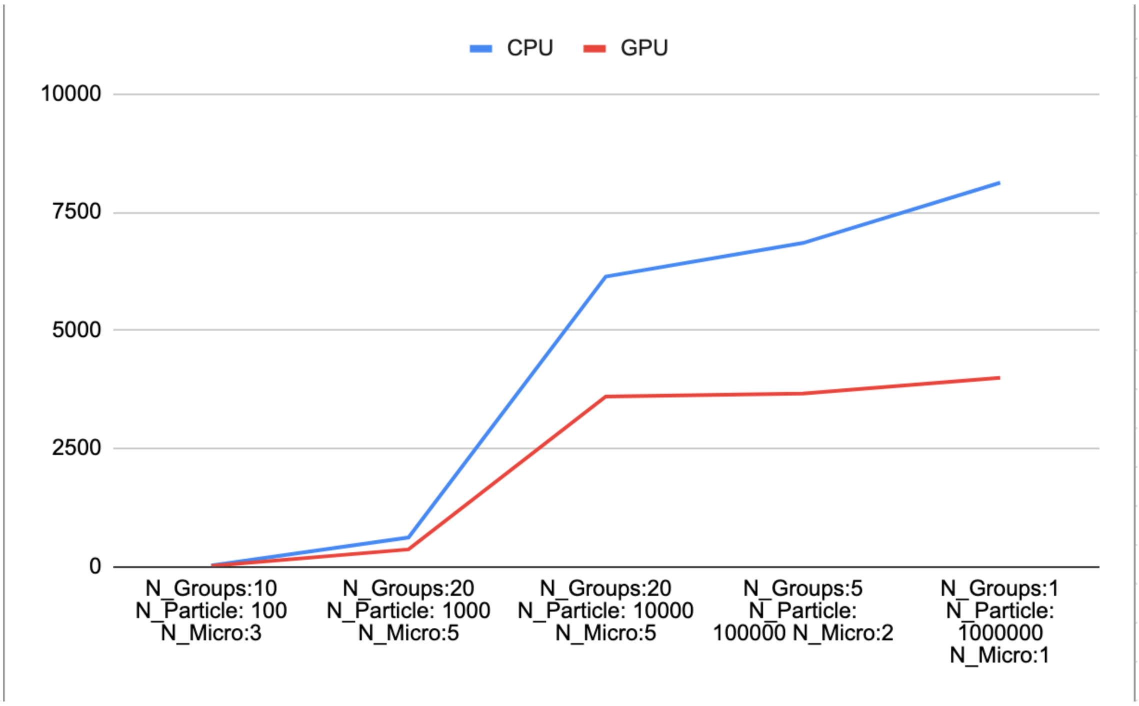


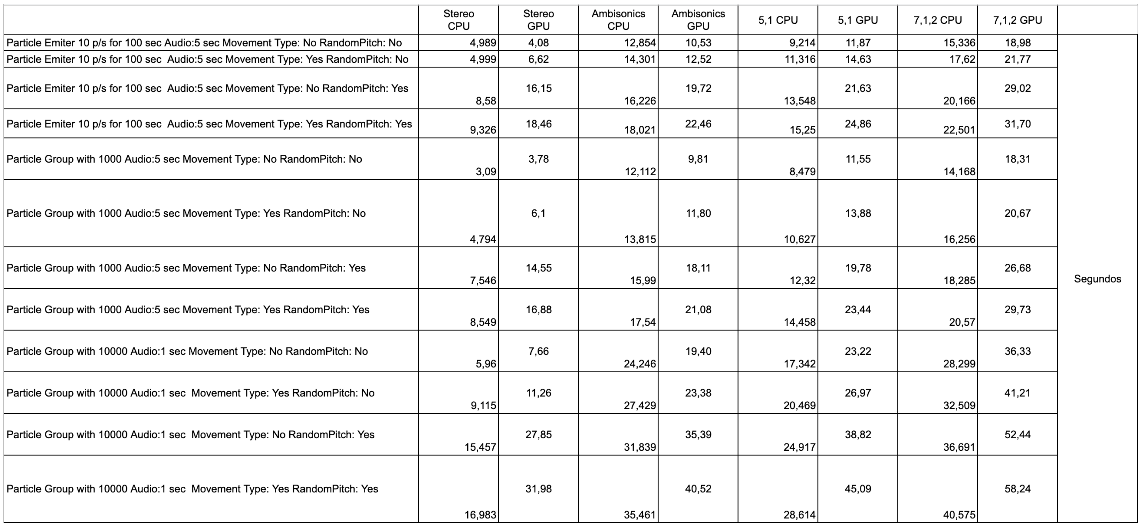
Figure 6.1: 1st Semester Results
Source: Author

waited for the command buffer to be completed, to check how much the system would increase if it happened, and it just increased in the order of the 8%. We can observe an increase in the thirteen versions of the GPU render time. In more detail, this was already justified in the implementation chapter, but it is because the sample size sent to the GPU was concise, creating a hefty overhead.
The final version, the eighteen, we can observe still is not the best, and the fifteen presented the best results due to the interpolation calculations not worth the migration to the GPU with the tests performed. The difference is not significant, and with some more optimizations, it could be passed the fifteen versions, but the new architecture implementation was chosen instead.
In the appendices **??** is a Google Sheet where I saved all the benchmarks and all the figures and graphs referred to in the previous paragraph.

6.2 Tests

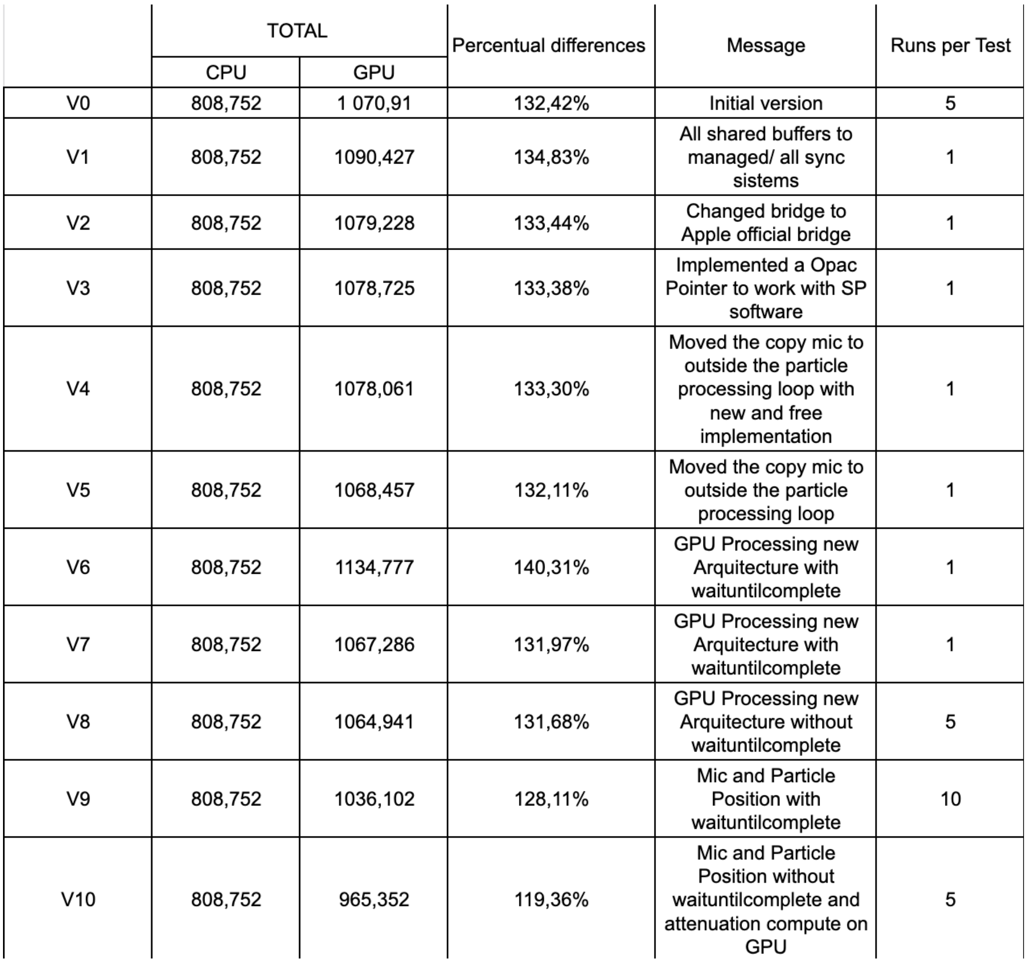
6.2.1 Functional Tests

The functionality tests were kept short since it only implemented integration testing, like checking if the device is detected in the system and if the application can create its main objects like the command queue, command Buffers, command encoders, etc.
In total, there were developed a total of w functionality tests since they were not the main focus of testing with GPU integration and were put to the side for the



	Stereo CPU	Stereo GPU	Ambisonics CPU	Ambisonics GPU	5,1 CPU	5,1 GPU	7,1,2 CPU	7,1,2 GPU	
Particle Emiter 10 p/s for 100 sec Audio:5 sec Movement Type: No RandomPitch: No	4,989	4,08	12,854	10,53	9,214	11,87	15,336	18,98	Segundos
Particle Emiter 10 p/s for 100 sec Audio:5 sec Movement Type: Yes RandomPitch: No	4,999	6,62	14,301	12,52	11,316	14,63	17,62	21,77	
Particle Emiter 10 p/s for 100 sec Audio:5 sec Movement Type: No RandomPitch: Yes	8,58	16,15	16,226	19,72	13,548	21,63	20,166	29,02	
Particle Emiter 10 p/s for 100 sec Audio:5 sec Movement Type: Yes RandomPitch: Yes	9,326	18,46	18,021	22,46	15,25	24,86	22,501	31,70	
Particle Group with 1000 Audio:5 sec Movement Type: No RandomPitch: No	3,09	3,78	12,112	9,81	8,479	11,55	14,168	18,31	
Particle Group with 1000 Audio:5 sec Movement Type: Yes RandomPitch: No	4,794	6,1	13,815	11,80	10,627	13,88	16,256	20,67	
Particle Group with 1000 Audio:5 sec Movement Type: No RandomPitch: Yes	7,546	14,55	15,99	18,11	12,32	19,78	18,285	26,68	
Particle Group with 1000 Audio:5 sec Movement Type: Yes RandomPitch: Yes	8,549	16,88	17,54	21,08	14,458	23,44	20,57	29,73	
Particle Group with 10000 Audio:1 sec Movement Type: No RandomPitch: No	5,96	7,66	24,246	19,40	17,342	23,22	28,299	36,33	
Particle Group with 10000 Audio:1 sec Movement Type: Yes RandomPitch: No	9,115	11,26	27,429	23,38	20,469	26,97	32,509	41,21	
Particle Group with 10000 Audio:1 sec Movement Type: No RandomPitch: Yes	15,457	27,85	31,839	35,39	24,917	38,82	36,691	52,44	
Particle Group with 10000 Audio:1 sec Movement Type: Yes RandomPitch: Yes	16,983	31,98	35,461	40,52	28,614	45,09	40,575	58,24	

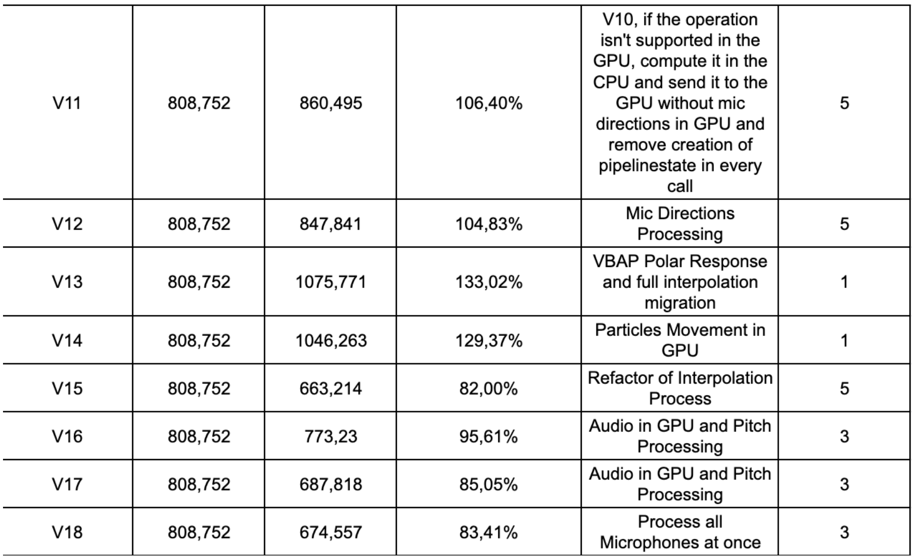
Figure 6.2: Baseline Tests
Source: Author



	TOTAL		Percentual differences	Message	Runs per Test
	CPU	GPU			
V0	808,752	1 070,91	132,42%	Initial version	5
V1	808,752	1090,427	134,83%	All shared buffers to managed/ all sync sistems	1
V2	808,752	1079,228	133,44%	Changed bridge to Apple official bridge	1
V3	808,752	1078,725	133,38%	Implemented a Opac Pointer to work with SP software	1
V4	808,752	1078,061	133,30%	Moved the copy mic to outside the particle processing loop with new and free implementation	1
V5	808,752	1068,457	132,11%	Moved the copy mic to outside the particle processing loop	1
V6	808,752	1134,777	140,31%	GPU Processing new Arquitecture with waituntilcomplete	1
V7	808,752	1067,286	131,97%	GPU Processing new Arquitecture with waituntilcomplete	1
V8	808,752	1064,941	131,68%	GPU Processing new Arquitecture without waituntilcomplete	5
V9	808,752	1036,102	128,11%	Mic and Particle Position with waituntilcomplete	10
V10	808,752	965,352	119,36%	Mic and Particle Position without waituntilcomplete and attenuation compute on GPU	5

Figure 6.3: Description table of the part of rendering with its render times
Source: Author

last task to be completed.



V11	808,752	860,495	106,40%	V10, if the operation isn't supported in the GPU, compute it in the CPU and send it to the GPU without mic directions in GPU and remove creation of pipelinestate in every call	5
V12	808,752	847,841	104,83%	Mic Directions Processing	5
V13	808,752	1075,771	133,02%	VBAP Polar Response and full interpolation migration	1
V14	808,752	1046,263	129,37%	Particles Movement in GPU	1
V15	808,752	663,214	82,00%	Refactor of Interpolation Process	5
V16	808,752	773,23	95,61%	Audio in GPU and Pitch Processing	3
V17	808,752	687,818	85,05%	Audio in GPU and Pitch Processing	3
V18	808,752	674,557	83,41%	Process all Microphones at once	3

Figure 6.4: Second part of description table of the part of rendering with its render times

Source: Author

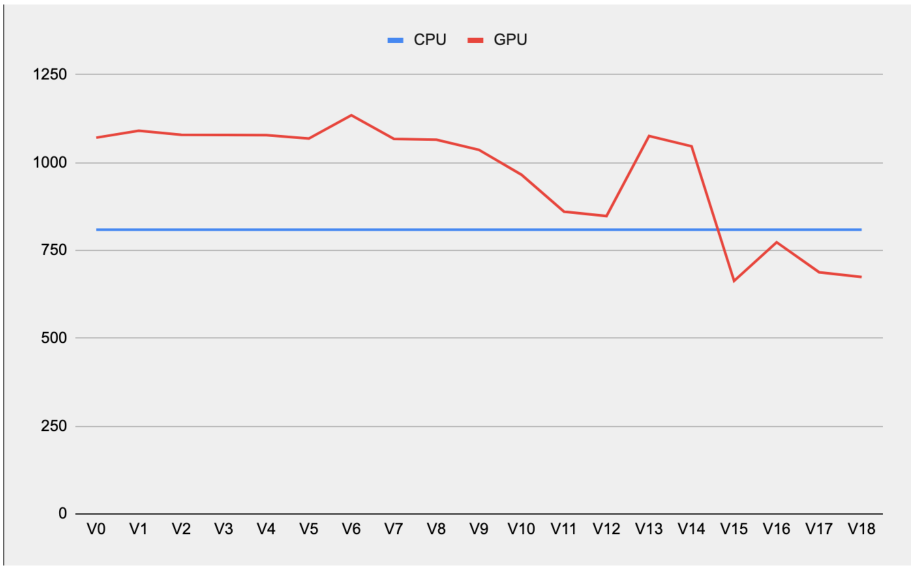


Figure 6.5: Graphic of the GPU and CPU render duration
Source: Author

6.2.2 Quality Tests

The quality tests were the main focus of the testing part of the internship, since one of the objectives of the internship was to migrate the render engine to the GPU and still deliver the same sound fidelity.

Due to Metal API not supporting double precision values for its operations, the testing needed a little deviation error acceptance. So in order to thoroughly test the render system, there was a need to test all paths possible and verify its outcome.

The first tests were mainly focused on testing all types of particle group renders process for all microphone systems types possible to be simulated in the

Sound Particles software, the monotype, the ambisonics type, and the multichannel type. So to test all these combinations, a bunch of tests was created that only changed the microphone type or the microphone type.
It was not developed all the heavy testing of the GPU, checking all the paths that were fragile in the system like changing the system properties, like the microphone or the group type or the buffer size, during the live render, closing the application suddenly, removing the graphics card unexpectedly from the computer, changing GPU device at render process, etc.
So, in total, there were developed twenty-nine tests to test the new render engine that tested different kinds of microphones system with particle groups of different dimensions or particle emitters.
It was also required to test if the new GPU render architecture adapts to the changes in the system, which means that if a sound is added or removed to the system, a buffer with its sound is created or destroyed, respectively if a particle group suffers a change in type the system should follow it and adapt to its change.

Chapter 7

Conclusions and Future Work

So far, the development provided a better knowledge of how to program in Graphical Processing Unit (GPU) and their programming directives like handling the GPU memory and ending the commands that the GPU need to execute the kernel shader functions.
The state of the art provided a better understanding of the current technologies/Application Programming Interface (API) that exist to program in the GPU and all the use cases that exist in the world that utilize it to deliver better performances and better parallelism between the Central Processing Unit (CPU) and the GPU, while the GPU is handling significant data operations the CPU is handling all the user interface processing, coordinate the GPU work and other usual jobs .
The experiences and the prototype development represent a big fundament in the document's writing since it provided a better knowledge of all requirements and processing that will need to exist to utilize the GPU to make all the computations that I wanted to perform.
At an initial stage, the future work will be to handle multiple GPUs to compute assignments since the Metal API provides a digital barrier between all different GPU devices if the duration of the internship will not be able to perform this stage of work.
The work developed, in my opinion, was a success considering its serious difficulty of implementation and since there is not in the current market any gpu render engine system so, for an internship concluding a prototype that provided better results than the CPU.
The results, in my opinion, could achieve better results if the new prototype was concluded and could dramatically increase the render velocity.

References

Aaron Klotz. Intel confirms poor arc gpu dx11 performance is a work in progress. `https://www.tomshardware.com/news/poor-dx11-performance-arc-gpus-constant-work-in-progress`, 2022. [Online; accessed 3-September-2022].

Yassmin Abdallah, Abdelaziz Abdelhamid, Taha Elarif, and Abdel-Badeeh M. Salem. Comparison between opengl es and metal api in medical volume visualisation. In *2015 IEEE Seventh International Conference on Intelligent Computing and Information Systems (ICICIS)*, pages 156–160, 2015. doi: 10.1109/IntelCIS.2015.7397213.

Anandtech. Opengl 4.4, opencl 2.0, opencl 1.2 spir announced. `https://www.anandtech.com/show/7161/khronos-siggraph-2013-opengl-44-opencl-20-opencl-12-spir-announced/2`. [Online; accessed 4-September-2022].

Apple. Calculating threadgroup and grid sizes. `https://developer.apple.com/documentation/metal/compute_passes/calculating_threadgroup_and_grid_sizes?language=objc`, a. [Online; accessed 4-September-2022].

Apple. Instance method addcompletedhandler:. `https://developer.apple.com/documentation/metal/mtlcommandbuffer/1442997-addcompletedhandler?language=objc`, b. [Online; accessed 4-September-2022].

Ebubekir Buber and Banu Diri. Performance analysis and cpu vs gpu comparison for deep learning. pages 1–6, 10 2018. doi: 10.1109/CEIT.2018.8751930.

Steven Chien, Ivy Peng, and Stefano Markidis. Performance evaluation of advanced features in cuda unified memory. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 50–57, 2019. doi: 10.1109/MCHPC49590.2019.00014.

Robert Fleming. General purpose programming on modern graphics hardware. volume 1, pages 1–5, 2008. doi: 10.1109/i-PACT44901.2019.8960106.

Alcides Fonseca and Bruno Cabral. Prototyping a gpgpu neural network for deep-learning big data analysis. *Big Data Research*, 8:50–56, 2017.

Gpuopen. V-ez brings "easy mode" to vulkan. `https://gpuopen.com/v-ez-brings-easy-mode-vulkan/`, 2018. [Online; accessed 4-September-2022].

Cecilia JM Hernandez M, Guerrero GD. Accelerating fibre orientation estimation from diffusion weighted magnetic resonance imaging using gpus. 2013.

Karl Rupp. Cpu, gpu and mic hardware characteristics over time. https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/, Edit, August 18th, 2016. [Online; accessed 21-January-2022].

E. Larsen and David McAllister. Fast matrix multiplies using graphics hardware. pages 43– 43, 12 2001. ISBN 1-58113-293-X. doi: 10.1109/SC.2001.10049.

Noel Lopes and Bernardete Ribeiro. GPUMLib: An Efficient Open-Source GPU Machine Learning Library. pages 355–362, 2011.

Michael Lujan, Michael Baum, Dayuan Chen, and Ziliang Zong. Evaluating the performance and energy efficiency of opengl and vulkan on a graphics rendering server. In *2019 International Conference on Computing, Networking and Communications (ICNC)*, pages 777–781, 2019. doi: 10.1109/ICCNC.2019.8685588.

Malcolm Owen. Compared: M1 vs m1 pro and m1 max. https://appleinsider.com/articles/21/10/30/compared-m1-vs-m1-pro-and-m1-max, Oct 30, 2021. [Online; accessed 17-January-2022].

NVIDIA. What is cuda? https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/, 2012. [Online; accessed 4-September-2022].

Olena. Gpu vs cpu computing: What to choose? https://medium.com/altumea/gpu-vs-cpu-computing-what-to-choose-a9788a2370c4, Feb 8, 2018. [Online; accessed 18-January-2022].

Overclock. Directx 12 / direct3d 12 feature levels and resources. https://www.overclock.net/threads/directx-12-direct3d-12-feature-levels-and-resources.1567968/, 2015. [Online; accessed 5-September-2022].

Meng Qi, Ke Yan, and Yuanjie Zheng. Gpredicates: Gpu implementation of robust and adaptive floating-point predicates for computational geometry. *IEEE Access*, 7:60868–60876, 2019. doi: 10.1109/ACCESS.2019.2911641.

SHOBHIT SETH, reviewed by ERIKA RASURE. Gpu usage in cryptocurrency mining. https://www.investopedia.com/tech/gpu-cryptocurrency-mining/, updated at 25 of August in 2021. [Online; accessed 20-January-2022].

Sound Particles. Sound particles. https://www.soundparticles.com. [Online; accessed 21-January-2022].

Telsa. Artificial intelligence and autopilot. https://www.tesla.com/AI. [Online; accessed 18-January-2022].

VMWare. Exploring the gpu architecture. https://core.vmware.com/resource/exploring-gpu-architecture#section1, 2022. [Online; accessed 21-January-2022].

Wikipedia contributors. Directcompute — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=DirectCompute&oldid=895110029, 2019. [Online; accessed 20-January-2022].

Wikipedia contributors. Apple m1 — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Apple_M1&oldid=1064052900, 2022a. [Online; accessed 17-January-2022].

Wikipedia contributors. General-purpose computing on graphics processing units — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=General-purpose_computing_on_graphics_processing_units&oldid=1066876703, 2022b. [Online; accessed 21-January-2022].

Wikipedia contributors. Opengl — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=OpenGL&oldid=1108194788, 2022c. [Online; accessed 4-September-2022].

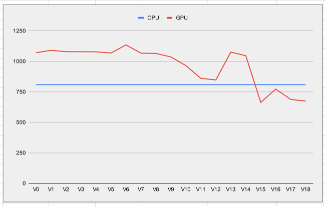
Junfeng Zhu, Gang Chen, and Baifeng Wu. Gpgpu memory estimation and optimization targeting opencl architecture. In *2012 IEEE International Conference on Cluster Computing*, pages 449–458, 2012. doi: 10.1109/CLUSTER.2012.9.

Appendices

Appendix A

	Mic_Type								
	Stereo CPU	Stereo GPU	Ambisonics CPU	Ambisonics GPU	5,1 CPU	5,1 GPU	7,1,2 CPU	7,1,2 GPU	
Particle Emiter 10 p/s for 100 sec Audio:5 sec Movement Type: No RandomPitch: No	4,989	4,08	12,854	10,53	9,214	11,87	15,336	18,98	Segundos
Particle Emiter 10 p/s for 100 sec Audio:5 sec Movement Type: Yes RandomPitch: No	4,999	6,62	14,301	12,52	11,316	14,63	17,62	21,77	
Particle Emiter 10 p/s for 100 sec Audio:5 sec Movement Type: No RandomPitch: Yes	8,58	16,15	16,226	19,72	13,548	21,63	20,166	29,02	
Particle Emiter 10 p/s for 100 sec Audio:5 sec Movement Type: Yes RandomPitch: Yes	9,326	18,46	18,021	22,46	15,25	24,86	22,501	31,70	
Particle Group with 1000 Audio:5 sec Movement Type: No RandomPitch: No	3,09	3,78	12,112	9,81	8,479	11,55	14,168	18,31	
Particle Group with 1000 Audio:5 sec Movement Type: Yes RandomPitch: No	4,794	6,1	13,815	11,80	10,627	13,88	16,256	20,67	
Particle Group with 1000 Audio:5 sec Movement Type: No RandomPitch: Yes	7,546	14,55	15,99	18,11	12,32	19,78	18,285	26,68	
Particle Group with 1000 Audio:5 sec Movement Type: Yes RandomPitch: Yes	8,549	16,88	17,54	21,08	14,458	23,44	20,57	29,73	
Particle Group with 10000 Audio:1 sec Movement Type: No RandomPitch: No	5,96	7,66	24,246	19,40	17,342	23,22	28,299	36,33	
Particle Group with 10000 Audio:1 sec Movement Type: Yes RandomPitch: No	9,115	11,26	27,429	23,38	20,469	26,97	32,509	41,21	
Particle Group with 10000 Audio:1 sec Movement Type: No RandomPitch: Yes	15,457	27,85	31,839	35,39	24,917	38,82	36,691	52,44	
Particle Group with 10000 Audio:1 sec Movement Type: Yes RandomPitch: Yes	16,983	31,98	35,461	40,52	28,614	45,09	40,575	58,24	
V1		4,067		10,986		12,559		19,715	
		6,591		13,539		15,089		22,148	
		16,333		20,12		21,94		29,084	
		18,353		22,55		24,989		32,151	
		3,705		10,117		11,88		19,057	
		6,157		12,287		14,128		21,45	
		14,83		18,038		21,178		26,964	
		16,828		21,163		23,726		30,325	
		7,821		19,802		25,166		37,784	
		12,106		24,024		27,51		41,765	
		28,012		35,569		38,705		54,007	
		31,974		40,545		44,532		59,058	
V2		4,213		10,699		12,246		19,247	
		6,828		13,339		14,811		21,82	
		16,255		19,77		22,05		28,563	
		18,785		22,401		24,083		31,889	
		3,667		10,003		11,749		18,451	
		6,033		12,218		14,125		20,872	
		14,959		18,53		20,325		26,735	
		17,049		21,159		23		30,105	
		7,968		19,456		24,7		36,893	
		12,242		23,699		27,083		40,906	
		27,991		35,198		39,002		53,482	
		32,017		40,213		44,275		58,124	
V3		4,199		10,682		12,22		19,189	
		6,784		13,311		14,782		21,668	
		16,271		19,891		21,798		29,136	
		19,053		22,352		24,494		32,026	
		4,663		10,003		11,848		18,41	
		6,157		12,264		14,472		20,87	
		15,204		18,391		20,251		27,087	
		16,963		20,375		23,182		30,223	
		7,91		19,27		24,713		36,832	
		12,041		23,603		26,881		40,665	
		27,852		35,31		38,173		52,745	
		31,875		40,221		44,213		58,202	
V4		4,255		10,847		12,365		19,266	
		6,758		13,46		14,944		21,796	
		16,744		20,124		22,872		28,872	
		18,37		22,55		25,247		31,704	
		3,682		10,028		11,688		18,582	
		6,09		12,109		13,82		20,975	
		14,53		18,37		19,955		26,73	
		16,654		21,081		23,029		30,052	
		7,593		19,472		24,643		37,137	
		11,888		23,701		27,48		40,932	
		27,596		35,142		38,32		53,063	
		31,927		39,887		43,811		57,92	
		4,018		10,693		12,2		19,054	
		6,541		13,331		14,745		21,586	
		16,091		20		21,513		28,609	

		TOTAL		Percentual differences	Message	Runs per Test
		CPU	GPU			
	V0	808,752	1 070,91	132,42%	Initial version	5
	V1	808,752	1090,427	134,83%	All shared buffers to managed/ all sync sistems	1
	V2	808,752	1079,228	133,44%	Changed bridge to Apple official bridge	1
	V3	808,752	1078,725	133,38%	Implemented a Opac Pointer to work with SP software	1
	V4	808,752	1078,061	133,30%	Moved the copy mic to outside the particle processing loop with new and free implementation	1
	V5	808,752	1068,457	132,11%	Moved the copy mic to outside the particle processing loop	1
	V6	808,752	1134,777	140,31%	GPU Processing new Arquitecture with waituntilcomplete	1
	V7	808,752	1067,286	131,97%	GPU Processing new Arquitecture with waituntilcomplete	1
	V8	808,752	1064,941	131,68%	GPU Processing new Arquitecture without waituntilcomplete	5
	V9	808,752	1036,102	128,11%	Mic and Particle Position with waituntilcomplete	10
	V10	808,752	965,352	119,36%	Mic and Particle Position without waituntilcomplete and attenuation compute on GPU	5
	V11	808,752	860,495	106,40%	V10, if the operation isn't supported in the GPU, compute it in the CPU and send it to the GPU without mic directions in GPU and remove creation of pipelinestate in every call	5
	V12	808,752	847,841	104,83%	Mic Directions Processing	5
	V13	808,752	1075,771	133,02%	VBAP Polar Response and full interpolation migration	1
	V14	808,752	1046,263	129,37%	Particles Movement in GPU	1
23/06/22	V15	808,752	663,214	82,00%	Refactor of Interpolation Process	5
	V16	808,752	773,23	95,61%	Audio in GPU and Pitch Processing	3
	V17	808,752	687,818	85,05%	Audio in GPU and Pitch Processing	3
	V18	808,752	674,557	83,41%	Process all Microphones at once	3



	Lagrange 6-point, 5th-order optimal 32x	er Hermite z-form	No ifs b4 lagrange	no ifs and no getnext,,,,	No audioParticle	No Update Floats	Att and Inter in C	Diff1	Diff2	Diff3	Diff4	Diff5	Diff6
	5,509	5,378	5,324	5,374	5,325	5,335	2,775	0,131	0,185	0,135	0,184	0,174	2,603
	5,338	5,403	5,321	5,32	5,38	5,38	2,441	-0,065	0,017	0,018	-0,042	-0,042	2,962
	16,917	17,066	17,475	16,324	5,706	17,455	16,383	-0,149	-0,558	0,593	11,211	-0,538	0,683
	16,714	17,346	17,063	16,606	5,778	17,591	16,281	-0,632	-0,349	0,108	10,936	-0,877	1,065
	4,623	4,505	4,48	4,391	4,432	4,33	2,37	0,118	0,143	0,232	0,191	0,293	2,135
	4,479	4,477	4,431	4,396	4,452	4,462	2,223	0,002	0,048	0,083	0,027	0,017	2,254
	15,454	15,607	16,312	15,13	5,053	16,779	15,269	-0,153	-0,858	0,324	10,401	-1,325	0,338
SUM	69,034	69,782	70,406	67,541	36,126	71,332	57,742	-0,748	-1,372	1,493	32,908	-2,298	12,04

V5	18,564	22,467	24,522	31,773
	3,749	10,039	11,85	18,37
	6,112	12,042	14,005	20,708
	14,618	17,934	19,813	27,11
	16,514	20,954	22,815	29,241
	7,619	19,436	24,65	36,485
	12	23,41	26,931	40,371
	27,387	35,024	37,852	52,563
	31,572	40,009	43,645	57,922
V6	5,269	12,07	14,158	23,14
	7,174	14,321	16,319	25,074
	15,985	20,857	21,941	32,333
	18,256	23,425	26,308	34,415
	4,574	11,002	12,848	21,021
	6,586	12,865	14,767	23,194
	14,577	19,042	20,652	29,562
	16,559	20,646	23,761	31,696
	8,299	19,689	24,878	38,469
	12,177	24,253	27,558	41,966
	27,935	36,28	39,644	57,804
	31,733	40,867	45,94	62,888
V7	4,035	10,524	12,178	19,089
	6,563	13,095	14,811	21,504
	15,746	19,629	22,143	29,085
	18,345	22,427	24,486	32,817
	3,635	9,911	11,647	18,293
	5,966	11,965	13,965	20,749
	14,271	17,945	19,574	26,769
	16,508	20,836	23,511	29,411
	7,522	19,013	24,451	36,534
	11,697	23,245	26,917	40,383
	27,312	34,707	38,164	52,807
	31,644	39,734	44,04	57,683
V8	4,078	10,352	12,155	18,862
	6,799	12,546	14,711	21,866
	16,1	19,695	21,364	28,72
	18,375	22,491	25,194	31,498
	3,833	9,577	11,546	18,316
	6,175	11,845	13,983	20,791
	14,358	18,054	19,705	26,539
	16,647	20,832	22,97	29,722
	7,573	18,967	23,436	36,231
	11,364	23,135	27,283	40,889
	27,419	35,046	38,612	52,05
	31,721	39,926	44,113	57,477
V9	4,078	10,352	12,155	18,862
	6,799	12,546	14,711	21,866
	16,1	19,695	21,364	28,72
	18,375	22,491	25,194	31,498
	5,029	10,413	11,763	13,034
	3,833	9,577	11,546	18,316
	15,162	19,525	20,663	21,285
	16,647	20,832	22,97	29,722
	8,768	20,156	20,732	25,676
	11,364	23,135	27,283	40,889
	29,328	37,98	40,509	41,922
	31,721	39,926	44,113	57,477
V10	4,078	10,352	12,155	18,862
	6,799	12,546	14,711	21,866
	16,1	19,695	21,364	28,72
	18,375	22,491	25,194	31,498
	2,856	5,105	6,5	11,303
	3,833	9,577	11,546	18,316
	14,197	15,87	16,284	20,415
	16,647	20,832	22,97	29,722
	5,401	10,059	12,825	21,695
	11,364	23,135	27,283	40,889
	26,94	30,184	31,631	39,93
	31,721	39,926	44,113	57,477
V11	3,49	6,319	8,911	14,997
	6,036	8,883	11,644	17,644
	15,883	17,634	19,263	24,866
	18,37	20,286	22,481	28,041
	2,843	5,15	6,12	10,725
	5,175	7,694	10,341	14,551
	14,297	15,931	16,518	20,563
	16,746	18,863	20,47	25,177
	5,596	10,353	11,771	20,954
	9,797	14,087	19,316	28,162
	27,295	30,608	31,342	40,095
	31,483	35,568	39,187	48,969
V12	3,644	6,268	8,864	14,309
	6,139	8,754	11,478	16,976
	15,512	17,678	19,204	24,16
	17,691	20,025	22,211	27,068
	2,778	5,257	6,003	10,526
	5,278	7,53	10,112	14,215
	13,969	15,78	16,286	20,226
	16,23	18,543	20,363	24,915
	5,441	10,08	11,82	20,553
	9,616	14,105	19,384	27,476
	26,495	30,742	31,1	39,242
	30,927	35,653	39,113	48,102
	4,878	9,258	11,829	20,842
	6,303	10,833	13,828	21,921

V13	15,581	19,369	21,808	31,357
	18,02	22,36	25,111	33,079
	4,629	9,028	11,715	29,278
	5,768	9,781	12,575	19,739
	13,348	16,952	18,851	32,649
	17,077	20,92	23,86	30,42
	8,209	16,168	21,196	53,456
	10,027	15,315	21,228	33,541
	26,24	32,197	35,935	63,672
	32,463	39,115	44,972	59,07
V14	5,62	10,672	13,218	27,327
	5,741	10,698	13,412	27,294
	15,32	17,86	20,5	36,557
	15,475	17,836	20,61	33,635
	4,662	8,911	11,211	24,078
	4,663	9,018	11,184	24,216
	13,95	17,069	19,208	30,736
	13,878	17,12	18,992	29,986
	9,253	16,183	20,293	45,224
	8,334	16,04	20,69	45,071
	26,34	32,793	36,622	64,187
	26,405	33,013	36,819	58,339
V15	2,484	4,629	5,385	9,548
	2,477	4,628	5,341	9,55
	16,369	17,87	18,118	21,21
	16,278	17,82	18,187	21,029
	1,789	3,349	3,687	6,51
	1,798	3,364	3,73	6,493
	14,862	16,476	16,764	19,432
	14,92	16,603	16,733	19,267
	2,977	5,691	6,116	10,984
	2,971	5,677	6,142	11,004
	28,06	31,143	31,473	36,769
	28,129	31,047	31,804	36,527
V16	2,493	4,589	7,574	22,297
	2,497	4,583	7,575	22,302
	7,8	11,965	14,412	29,676
	7,959	12,196	14,884	29,856
	1,803	3,302	6,246	19,76
	1,8	3,298	6,223	19,619
	17,086	24,626	27,479	40,555
	17,156	28,041	27,008	39,862
	2,991	5,621	11,114	36,575
	2,986	5,624	11,097	36,564
	8,427	11,796	18,643	48,079
	8,465	11,78	18,645	48,301
V17	2,48	4,468	7,565	22,079
	2,488	4,454	7,568	22,032
	3,405	5,446	8,946	24,567
	3,386	5,449	8,892	24,567
	1,785	3,129	6,237	19,686
	1,761	3,119	6,213	19,782
	16,025	19,028	22,092	36,852
	16,097	18,956	22,154	36,995
	2,949	5,183	11,055	36,516
	2,945	5,172	11,039	36,593
	7,948	10,928	17,865	47,4
	7,967	10,948	17,871	47,736
V18	2,167	3,54	6,871	20,36
	2,178	3,547	6,85	20,286
	3,131	4,612	8,505	22,766
	3,132	4,591	8,449	22,779
	1,765	3,121	6,512	19,522
	1,749	3,094	6,492	19,495
	16,134	19,048	22,297	36,486
	16,075	19,161	22,341	36,265
	2,904	5,194	11,681	36,21
	2,908	5,182	11,75	36,154
	7,971	10,939	18,674	47,205
	7,967	10,951	18,659	46,887