

Parallele Signalverarbeitung mit CUDA

Fouriertransformationen und positionsbezogene Audiomanipulation

MASTERARBEIT

zur Erlangung des akademischen Grades
eines Master of Science



 **Universität Trier**

Fachbereich IV

Lehrstuhl für

Systemsoftware und Verteilte Systeme

Erstprüfer: Prof. Dr. Peter Sturm

Zweitprüfer: Prof. Dr. Benjamin Weyers

Fabian Hasselbach

Eutscheider Straße 21, 51570 Windeck

Matrikelnr: 1106363

Windeck, 11. November 2021

Inhaltsverzeichnis

1	Einleitung	1
2	Idee und Zielsetzung	2
3	Signalverarbeitung	4
3.1	WAV Format [4] [5]	4
3.2	Nyquist	5
3.3	Fouriertransformation	7
4	CUDA	9
4.1	Was ist CUDA	9
4.2	Vorteile und Anwendungsgebiete	9
4.3	Anwendungsfall NVIDIA GeForce GTX 1080	9
5	Teilprojekt: „Parallele Fourieranalyse mit CUDA“	10
5.1	Überblick	10
5.2	StreamWriter(C/C++)	11
5.3	Code Architektur	13
5.3.1	StreamWriter	14
5.3.2	AudioTracer	20
5.3.3	AudioTracer (Python)	33
5.3.4	Bokeh Server zur Livevisualisierung	36
6	Ergebnisse	39
6.1	AudioTracer C	40
6.2	AudioTracer Python	42
6.3	Vergleich der Testreihen	44
7	Teilprojekt: „Audioparcours mit CUDA“	45
7.1	Überblick	45
7.2	Code Architektur	49
7.3	Ergebnisse	57
8	Skalierbarkeit	58
9	Zusammenfassung	59
10	Ausblick	60
11	Eigenständigkeitserklärung	66

1 Einleitung

In einer digitalen Welt des immer größer werdenden Realismus in künstlichen Umgebungen werden aufgrund hoher Komplexität und der reinen Masse an Daten bereits Teilbereiche der Bildentwicklung über die Fähigkeit für hohe Parallelisierung von Grafikkarten verwirklicht. Beispielsweise ist die Technik des Raytracings und die damit verbundene Fähigkeit, realistische Schatten und Licht in künstlichen Umgebungen zu erzeugen in der Bildverarbeitung, der Gaming-Szene oder VR-Anwendungen heutzutage schon sehr weit erforscht. Die Echtheit von 3D-Anwendungen hat einen immer höheren Stellenwert für deren Benutzer. Gleichzeitig sticht jedoch heraus, dass die Entwickler solcher Anwendungen im Bereich von Audiorealismus eigene Schnittstellen realisieren, die z.B. positionsbezogene Audiowiedergabe zu simulieren versuchen. Im Bereich der Spielentwicklung gibt es sogar bei Triple-A-Titeln viele Beispiele, deren positionsbezogene Audioschnittstellen nicht sehr weit entwickelt sind. Auch in der Audioforschung wären solche Schnittstellen in vielen Bereichen wie Auditorische Szenenanalyse, Binauraltechnik, Hörtechnik in der biomedizinischen Technik, Lärmforschung und Psychoakustik [1] brauchbar. Aus einem Bericht von TECHPOWERUP [2] vom 16.08.2018 ist zu entnehmen, dass der Grafikkarten-Hersteller Nvidia in der RTX-Reihe seiner Produkte eine solche Schnittstelle für positionsbezogenes Audio etablieren will, indem angenommen wird, dass sich Soundquellen analog zu Lichtquellen auf unterschiedlichen Oberflächen verschieden verhalten. Der Entwicklungsstand dieser Schnittstelle ist bis jetzt nicht bekannt.

Diese Abschlussarbeit beschäftigt sich mit dem Thema paralleler Signalverarbeitung in Form von paralleler Anwendung von Fouriertransformationen im ersten und positionsbezogenem Audio im zweiten Teilprojekt. Es wird versucht eine initiale Herangehensweise zu liefern, mit der solche Schnittstelle in Zukunft realisiert werden können. Die Implementierung eines solch umfassenden Themenbereichs übersteigt den Umfang dieser Abschlussarbeit immens. Weitere zukünftige Forschungsarbeiten wären nötig, um dieses Thema weiter voranzutreiben. Die hier implementierten Techniken beschäftigen sich zum einen mit der parallelen Extraktion von anteiligen harmonischen Frequenzen einer Audiodatei und deren Visualisierung über Fouriertransformationen. Zum anderen wird eine Möglichkeit geboten, eine laufende Wiedergabe eines hier sogenannten Audioparcours zu manipulieren, wodurch eine positionsbezogene Simulation von Audio erreicht wird.

Der Quellcode, auf dem diese Abschlussarbeit, liegt auf einem Github-Repository [3] unter folgendem Link: https://github.com/fahaba/Master_Thesis.

2 Idee und Zielsetzung

Der Grundlegende Gedanke zur Durchführung dieser Arbeit ist, den Bereich des Parallel Computings auf die Signalverarbeitung anzuwenden. In der Anwendung werden speziell Audiodateien im WAV-Dateiformat dazu herangezogen. In diesen WAV-Dateien sind Audiosignale für jeden Audiokanal gespeichert, was bei einer Erhöhung der Audiokanäle dazu führt, dass die Anzahl der gespeicherten Daten in Relation zu diesen vervielfacht wird. Die Wiedergabe dieser Audiodateien wird sehr wahrscheinlich wenig problematisch sein und auch bleiben. Diese Arbeit setzt jedoch bei der Analyse solcher Dateien an. Die Extraktion der anteiligen Tonfrequenzen ist in der Signalverarbeitung das zentrale Werkzeug zur weiteren Verarbeitung von Audiodateien. Dieses Verfahren wird durch die Fourieranalyse bzw. Fouriertransformation realisiert und in Kapitel 3 näher thematisiert. Im Grunde kann damit über komplexe mathematische Strukturen errechnet werden, aus welchen harmonischen Signalen mit unterschiedlicher Amplitude, Frequenz und Phase ein periodisches Signal zusammengesetzt ist. Das Ziel ist es, diese mathematischen Berechnungen auf die Shader-Kerne der Grafikkarte zu verteilen und parallel auszuführen, um damit eine Leistungssteigerung bzw. Performance-Steigerung zu erreichen. Herkömmliche CPUs reichen technologiebedingt oft nicht mehr aus, um viele kleine Problemstellungen effizient und vor allem parallel zu berechnen. Die Parallelisierbarkeit ist auf die Anzahl der Threads einer CPU begrenzt. Im Beispiel des Anwendungsfalles wird ein Intel I7 7700k verwendet. Die technischen Spezifikationen geben vier Kerne an, wobei durch Hyperthreading Technologie acht logische Kerne realisiert werden. Somit können bis zu acht Rechenoperationen parallel ausgeführt werden. Diese Tatsache wird als Problemstellung angenommen und an dieser Stelle wird das Ziel definiert, dass statt wenig große Rechenoperationen im Bezug auf die Signalverarbeitung die nötigen Berechnungen auf viele kleinere Teilproblemstellungen heruntergebrochen werden, um diese mithilfe der Grafikkarte zu parallelisieren. Die Grafikkarte bietet durch frei programmierbare Shader-Kerne über Schnittstellen wie CUDA (Nvidia) oder OpenCL(AMD) die Möglichkeit, solche Aufgaben effizient zu lösen. Tausende solcher Kerne mit geteiltem Speicher (RAM) sind netzförmig auf einer GPU verbaut. Auf fundamentaler Ebene dienen sie dem Zweck, grafische Informationen in Form von Pixelinformationen oder Polygonen zu manipulieren bzw. auszuwerten. Somit ist die Hauptaufgabe einer Grafikkarte das Lösen von unzähligen Gleichungssystemen und anderen mathematischen Operationen. Die ALUs (Arithmetic Logic Unit) übernehmen somit den fundamentalen Baustein einer Grafikkarte.

Eine naheliegende „Zweckentfremdung“ der Shader-Kerne ist die Ausführung von individuellem Code basierend auf der Lösung von rechenintensiven Problemen, die sich auf viele kleine Teilprobleme zerlegen lassen. Im Anwendungsfall der Audiosignalverarbeitung ist eine solche Situation gegeben. Das Grundproblem besteht darin, aus den Audiodaten einer WAV-Datei mithilfe der Verwendung von Fourier Transformationen die anteiligen Tonfrequenzen zu berechnen und digital darzustellen. Die Aufteilung des zu untersuchenden Bereichs einer Tonspur auf die CUDA-Kerne wird von der CPU

bzw. dem Host-System durchgeführt. Jedem Kern wird von diesem Bereich eine Menge an Audiosamples zugeordnet und über die Fouriertransformation die Stärke des Auftretens der zugeteilten Frequenz berechnet. Durch die Zusammenarbeit von CPU und Shader-Kernen der Grafikkarte entsteht somit ein heterogenes System zur Lösung der o.g. Problemstellung. Die errechneten anteiligen Frequenzen werden letztlich mithilfe einer TCP-Schnittstelle über einen Webserver zur Laufzeit visualisiert. Grundsätzlich thematisiert diese Arbeit ausschließlich das WAV-Format. Die Architektur des zugrundeliegenden Programms ist jedoch leicht auf andere Formate übertragbar.

Ein weiterer Teilbereich dieser Arbeit umfasst die Manipulation einer Audioquelle über die CUDA-Schnittstelle. Das Ziel ist es, einen variablen räumlichen Punkt zu wählen der als Quelle eines Audiosignals dient, um dann diese mithilfe der Shader-Kerne zu manipulieren, sodass über Nutzereingaben, die die relative Position zur Audioquelle verändern, ein positionsbezogener Effekt entsteht. Genauer werden Signalabschwächung und die räumliche „Lage“ des Signals mithilfe einer Matrix bzw. eines Koordinatensystems (CUDA-Grid) in Abhängigkeit der Position verrechnet bzw. manipuliert. Dadurch entstehen Übergangszustände von Daten, die wiederum die Eingabedaten für darauffolgende Iterationen darstellen. Die Quelle des Signals ist zur Laufzeit variabel, kann also durch bestimmte Eingaben verändert bzw. verschoben werden. Einzelheiten zu diesen Teilbereichen der Zielsetzung wird in den jeweiligen Kapiteln näher beschrieben.

3 Signalverarbeitung

3.1 WAV Format [4] [5]

Das Waveform-Audiodatei-Format WAV ist ein Audioformat, das zum Speichern von Audioinformationen als Bitstream dient. Dieses Audioformat basiert auf der von Microsoft entwickelten **Resource Interchange File Format Spezifikation** (RIFF), die die Speicherung von Audiostreams in Teilen bzw. Chunks vorsieht. Jede WAV-Datei besitzt eine Headerinformation, die in solche Chunks aufgeteilt wird, gefolgt von den eigentlich gespeicherten Daten. Diese Chunks werden durch eine eindeutige Kennzeichnung (ID), die Länge der darin befindliche Daten sowie durch die Daten selbst beschrieben. WAV-Dateien bestehen in der Regel aus den drei Chunks RIFF-Header, FMT-Subchunk und einem Subchunk für die Daten (DATA-Chunk).

Der RIFF-Header besteht aus einer 12 Byte Information und ist in jeweils 4 Byte unterteilt. Die ersten 4 Byte geben die ChunkID (In diesem Fall RIFF) in ASCII Form an. Ist die ChunkID mit RIFF angegeben, liegen die folgenden Daten im Little-Endian Format vor. Bei RIFX würde das Big-Endian Format angenommen. Die folgenden 4 Byte geben die gesamte Größe (ChunkSize) der WAV-Datei an (abgesehen von den beiden vorherigen 4 Byte Blöcken ChunkID und ChunkSize). Die folgenden 4 Byte beschreiben das Format, was im vorliegenden Fall von WAV-Dateien das Wort **WAVE** enthält.

Im FMT-Subchunk sind wichtige Metainformationen über die Audiodatei gespeichert. Vier Byte Subchunk1ID sind durch den String **FMT** beschrieben, gefolgt von der vier Byte großen **Subchunk1Size**. Im Falle des relevanten Audioformats PCM ist hier die Restgröße des FMT-Subchunks festgehalten. Byte 20 und 21 gibt das oben genannte Audioformat als Zahlenwert an. PCM wird durch eine 1 typisiert und beschreibt einen unkomprimierten Datenstrom in der Datei. Komprimierung würde durch andere Werte identifiziert. Die Bytes 22 und 23 geben die Anzahl der verwendeten Audiokanäle an (NumChannels). Dies ist eine sehr wichtige Information zur räumlichen Aufteilung der Audiosignale z.b. zur Wiedergabe auf Audiogeräte mit mehreren Ausgabekanälen. Die Sample Rate wird in den Bytes 24-27 angegeben und enthält die Angabe, wie viele Samples bei der Wiedergabe in einer Sekunde verarbeitet werden. Prinzipiell ist hier die Qualität der Audiodatei festgehalten. Oftmals liegen WAV-Dateien in der Frequenz 44100hz vor. Bytes 28-31 geben die Größe eines einzelnen Samples über alle Kanäle betrachtet an. Die folgenden zwei Bytes (34 und 35) enthalten die Größe für jeweils ein Sample, wobei sich hieraus auch die Kanalgröße in einem Sample durch BitsPerSample / NumChannels errechnen lässt. In dieser Arbeit wird ausschließlich mit 16 bit PCM WAV-Dateien gearbeitet.

Der DATA-Subchunk enthält die eigentliche Audioinformation im zuvor vorgestellten Format, beschrieben durch den FMT-Subchunk. Jeweils vier Bytes geben Subchunk2ID ("data") und die Größe dieses Chunks als Subchunk2Size an. Danach folgen die eigentlichen Daten.

Abbildung 1 zeigt diese Formatspezifikation anhand eines in dieser Arbeit verwendeten Audiofiles.

Chunk descriptior											
52	49	46	46	20	4F	66	02	57	41	56	45
R	I	F	F	ChunkSize = 40255264				W	A	V	E

FMT Subchunk I											
66	6D	74	20	10	00	00	00	01	00	02	00
f	m	t		ChunkSize = 16				Audioformat = 1 (PCM)		NumChannels = 2	

FMT Subchunk II											
44	AC	00	00	10	B1	02	00	04	00	10	00
SampleRate = 44100				ByteRate = 176400				BlockAlign = 4		BitsPerSample = 16	

Data Subchunk											
64	61	74	61	FC	3E	66	02	A5	FF	8E	FF
d	a	t	a	Subchunk2Size = 40255228				Sample 1 left channel		Sample 1 right channel	

Abbildung 1: Metainformationen des Audioformat WAV

3.2 Nyquist

Nyquist ist eine Programmiersprache zur synthetischen Erzeugung von Audiosignalen und zur Analyse von bestehenden Audioquellen. Basierend auf der Sprache Lisp stellt Nyquist eine Erweiterung des XLISP Dialekt dar. Die Entwicklung wurde von der Yamaha Corporation und IBM unterstützt vom Mitgründer des Audiotbearbeitungsprogramms Audacity Roger Dannenberg durchgeführt [6].

Audacity biete eine direkte Schnittstelle zur Nyquist Programmiersprache als verwendbares Plugin. In dieser Arbeit wird dieses Plugin benutzt, um Wind-ähnliche Audiospuren zu generieren, um diese im Weiteren mitunter analysieren zu können. In folgender Abbildung 2 wird das dazu verwendete Codesegment dargestellt.

In den ersten vier Zeilen werden Header-Informationen festgelegt, die Audacity auswerten kann. Der Typ gibt dabei an, unter welcher Rubrik (Generate, Effect, Analyze, etc.) das Plugin zu finden sein wird. In den Zeilen 6 bis 9 werden Benutzereingaben angefragt und deren Typ festgelegt. Die Eingabe wird als Schieberegler zugreifbar und durch die jeweils am Ende der Zeile befindlichen Werte begrenzt. Für das Wind Beispiel werden Dauer `dur`, Skalierung `scal`, Zyklen pro Sekunde `cps` und Bandbreite `bw` benötigt. Die Bandbreite gibt dabei die Amplitudengröße der Signalwellen an. In Zeilen 11 bis 22 wird die Funktion zur Generierung des Wind Beispiels definiert. Der funktionale Aufbau des Windbeispiels nimmt als Grundlage ein Muster aus dem offiziellen Audacity

```

1 ;nyquist plug-in
2 ;type generate
3 ;version 1
4 ;name "Wind"
5
6 ;control dur "Duration" float "duration" 0.1 0.0 360
7 ;control scal "scale" float "scale" 0.1 0.0 10.0
8 ;control cps "cps" float "cps" 1 0 2000
9 ;control bw "bw" float "bw" 1 0 100
10
11 define function buildPWL()
12     return pwl(0, 0.74, 0.2, 0.96, 0.5, 0.8, 0.75, 1.16, 0.9, 0.97, 1, 0.74, 1)
13
14 define function buildENV()
15     return env(0.07, 0.08, 0.1, 1, 0.6, 0.8, 1)
16
17 define function buildReson(scal, cps, bw, vpwl)
18     return reson(scale(scal, noise()), cps, bw, 2) +
19         reson(scale(scal * 1.13, noise()), cps * vpwl, bw * 1.042, 2)
20
21 define function wind(dur, scal, cps, bw)
22     return stretch(dur, buildENV() * buildReson(scal, cps, bw, buildPWL()))
23
24 return wind(dur, scal, cps, bw)

```

Abbildung 2: Nyquist synthetischer Wind

Forum [7]. Dieses basiert auf einer Transformation und verschiedenen Skalierungen des Weißen Rauschens, das in Audacity mit der Buildin Funktion `noise()` erzeugt werden kann. Eine detailliertere Beschreibung ist in der offiziellen Anleitung [8] zu finden.

Mit der Methode `wind` können letztendlich einzelne Windspuren mit einer festgelegten "Tonhöhe" erzeugt werden. Um ein komplexeres Beispiel zu erzeugen, wurden zwei Tonspuren des Windes auf jeweils einen Stereo-Kanal überlagert. Außerdem können in der Audacity Nyquist-Eingabeaufforderung Sinuswellen zu einer gegebenen Frequenz erzeugt werden. Zur Verifizierung der in Kapitel 5 implementierten Fouriertransformationen, wurde analog zu Abbildung 2 mit einem kleinen Nyquist Plugin Code: `return *track* + osc(freq)` eine Stereo WAV-Datei mit zwei Sinuswellen 30 bzw. 60 Hertz auf dem linken bzw. rechten Audiokanal erzeugt.

3.3 Fouriertransformation

Die Fouriertransformation bzw. Fourieranalyse ist einer der weit verbreitetsten Werkzeuge im Bereich der Signalanalyse und Signalverarbeitung. Basierend auf den Fourier Reihen (von Jean-Baptiste-Joseph de Fourier) kann damit errechnet werden, aus welchen harmonischen Signalen ein periodisches Signal aufgebaut ist. Dabei unterscheiden sich die Ausgangssignale zwischen Amplitude, Frequenz und Phasenlage bzw. Phasenverschiebung. Der Grundgedanke bei der Fourier Analyse ist, dass jedes periodisches Signal durch ein Gleichanteil und einer unendlichen Summe harmonischer Signale dargestellt werden kann. Harmonische Signale werden in Form von Linearkombinationen dargestellt. Die Bestimmung der Unbekannten, die in diesem Fall Fourierkoeffizienten genannt werden, ist Gegenstand der Fourieranalyse. Die allgemeine Fourierreihe oder auch trigonometrische Reihe wird dargestellt durch:

Definition (1): Fourier Reihe

$$f(t) = c_0 + \sum_{n=1}^{\infty} [a_n \cos(n\omega_0 t) + b_n \sin(n\omega_0 t)] \quad (1)$$

mit $\omega_n :=$ ganzzahliges Vielfaches einer Kreisfrequenz von ω_0
und $\omega_0 = 2\pi/T$ mit T Periodendauer.

DFT In Definition (1) ist die Zerlegung eines periodischen Signals als unendliche Summe von Linearkombinationen dargestellt. Ein Spezialfall dieser Formel, deren Koeffizienten in komplexer Form angegeben werden, ist in Definition (2) erkennbar und mithilfe von [9] nachzuvollziehen.

Definition (2): Diskrete Fouriertransformation

$$f(k) = 1/N \sum_{l=0}^{N-1} f(l) \cdot e^{\frac{-ikl \cdot 2\pi}{N}} \quad (2)$$

mit Zeilenindex k und Spaltenindex l

Der Zusammenhang zwischen Definitionen (1) und (2) wird klar, wenn das auf der Eulerschen Formel basierende Theorem von De Moivre [10] angewendet wird.

Definition (3): Herstellung des Zusammenhangs zw. (1) und (2)

$$\begin{aligned}
 f(k) &= 1/N \sum_{l=0}^{N-1} f(l) \cdot e^{\frac{-i k l \cdot 2\pi}{N}} \\
 &= 1/N \sum_{l=0}^{N-1} f(l) \cdot \left[\cos\left(\frac{2\pi}{N} k l\right) - i \cdot \sin\left(\frac{2\pi}{N} k l\right) \right]
 \end{aligned} \tag{3}$$

FFT Die Fast Fourier Transformation von James W. Cooley und John W. Tukey ist ein verbesserter Algorithmus zur Durchführung der Fouriertransformation. Analog zur Namensgebung werden wesentlich weniger Rechenoperationen benötigt, um das Ergebnis zu errechnen. Grundannahme ist eine Länge des Eingangssignals von $N = 2^M$, mit $M \in \mathbb{N}$. Die Besonderheit ist, dass der Algorithmus rekursiv ausgeführt wird. Gerade und ungerade Vektoren vom Ausgangsvektor \vec{f} werden rekursiv aufgeteilt, bis N Vektoren der Länge 2^0 vorliegen:

Definition (4): Aufteilung des Ausgangsvektors \vec{f}

$$\begin{aligned}
 \vec{f} &= f\left(2k \cdot \frac{2\pi}{N}\right) \\
 \vec{f} &= f\left([2k + 1] \cdot \frac{2\pi}{N}\right)
 \end{aligned} \tag{4}$$

Mithilfe der Diskreten Fouriertransformation f^{DFT} lässt sich die Gleichung der Fast Fouriertransformation aufstellen:

Definition (5): Fast Fourier Transformation [9]

$$\begin{aligned}
 f_k^{DFT} &= \frac{1}{2} \cdot (f_{2k}^{DFT} + [e^{-i2\pi/N}]^k \cdot f_{2k+1}^{DFT}) \\
 f_{k+\frac{N}{2}}^{DFT} &= \frac{1}{2} \cdot (f_{2k}^{DFT} - [e^{-i2\pi/N}]^k \cdot f_{2k+1}^{DFT})
 \end{aligned} \tag{5}$$

Randnotiz Zu beiden Varianten der Fourier Transformation existiert eine wohl definierte Inverse Rechenoperation, die aus den harmonischen Signalen wieder das periodische Ausgangssignal errechnet. Diese Inverse wird hier nicht weiter erläutert, da sie in dieser Arbeit keine Anwendung findet.

4 CUDA

4.1 Was ist CUDA

NVIDIA CUDA [11] ist eine Plattform zur parallelen Ausführung von Nutzer definiertem Code auf den Shader-Kernen der Grafikkarte. In bestimmten Anwendungsfällen kann die Ausführungsgeschwindigkeit von Applikationen dadurch stark verbessert werden. Grundsätzlich ist dies in solchen Applikationen möglich, deren Aufgabe in viele kleine Teilprobleme zerlegt werden kann. Den sequentiellen Anteil dieser Programme, wie z.B. die Bereitstellung der Eingabeinformationen und Verteilung dieser Informationen auf die verwendeten Shader-Kerne der Grafikkarte übernimmt in solchen Anwendungen die CPU, während die jeweiligen Teilprobleme auf den Shader-Kernen selbst ausgeführt wird. Außerdem wird die zentrale CPU dafür verwendet, die Ausgabe der entsprechenden GPU-Prozessen zu verwalten, zu sammeln und letztendlich auf verschiedene Art und Weisen darzustellen oder weiterzuleiten. Diese Technologie ermöglicht die Etablierung von neuronalen Netzen und Deep-Learning Strukturen, die heutzutage riesige Anwendungsgebiete umfassen. Letzteres wird in dieser Arbeit nicht thematisiert. Das von NVIDIA entwickelte CUDA-Toolkit ist kompatibel mit den meisten größeren Programmiersprachen (C, C++, C#, Java, Python).

4.2 Vorteile und Anwendungsgebiete

Die Vorteile, die eine Plattform wie CUDA bietet sind zahlreich. Allen voran und bereits erwähnt steht die Verarbeitung von vielen parallelen Rechenoperationen im Vordergrund. Komplexe Probleme können dadurch von der CPU auf die GPU ausgelagert werden, um einen Overhead auf der CPU zu vermeiden. Strukturen wie Neuronale Netze können im Deep-Learning Bereich etabliert werden, wobei mit einer großen Masse an Daten gearbeitet wird. Einen großen Sprung durch parallel Computing wurde im Bereich von KI-Realisierungen für autonome Prozesse wie autonomes Fahren, in der Robotik, der Auswertung von riesigen Datenmengen z.B. für Mustererkennung, Prognosen, u.v.m. erreicht. Diese aufgelisteten Techniken und Prozessstrukturen sind in der heutigen Welt nicht mehr vernachlässigbar und auch sinnvoll, da die Technologien der CPU mittlerweile an physikalische Möglichkeiten limitiert sind, während Grafikkarten immer größer skaliert werden können und auch im Verbund arbeiten können (SLI im Falle von Nvidia Produkten). Ein großes Anwendungsgebiet stellten z.B. digitale Währungen wie Bitcoin, Ethereum, etc. dar, die über Blockchain-Technologien auf der Grafikkarte erzeugt, bzw. berechnet werden können.

4.3 Anwendungsfall NVIDIA GeForce GTX 1080

Die verwendete Hardware, auf die sich diese Arbeit bezieht ist eine GEFORCE GTX 1080. NVIDIA gibt in den technischen Spezifikation [12] 2560 NVIDIA CUDA-Kerne mit einer Basistaktung von 1607 MHz und einem Boost-Takt von 1733 MHz an.

5 Teilprojekt: „Parallele Fourieranalyse mit CUDA“

5.1 Überblick

Um ein Gesamtbild über das implementierte Projekt zu bekommen, wird an dieser Stelle ein grober Rahmen, der die wichtigsten Funktionalitäten und Anforderungen an die Implementierung umfasst, angegeben. Aus dem Abschnitt Idee und Zielsetzung 2 ist bereits bekannt, dass Audiodaten in Form von WAV-Dateien analysiert und deren anteilige Frequenzen dargestellt werden, was in einer sequentiellen Umgebung zunächst kein größeres Problem darstellt. Diese Arbeit geht jedoch einen Schritt weiter, parallelisiert diesen Prozess und nutzt die Schnittstelle CUDA aus, um je nach verwendeter Hardware dieses Problem auf die Grafikkarte zu skalieren. Mit dem Schritt der Parallelisierung stellen sich folgende Fragen:

- Wie werden die Audiodaten auf die Shader-Kerne der Grafikkarte übertragen?
- Welche Form haben diese Daten?
- Wie und in welchen Intervallen werden diese Daten portioniert und wie viele Daten werden parallel verarbeitet?
- Mit welcher Methode werden die Daten verarbeitet bzw. umgerechnet?
- Wie werden die Daten nach erfolgreicher Berechnung wieder verfügbar gemacht?
- Form der Visualisierung?

Einige dieser Fragen bringen komplexere Strukturen mit sich und sind zunächst nicht einfach zu beantworten. Eine Audiodatei ist im Grunde genommen eine Textdatei, mit vielen Hexadezimalen Werten und einer Header-Information, wie in Abschnitt WAV-Format 3.1 beschrieben wurde. Aus dieser Datei muss die Information über Sample-Rate, Anzahl der Kanäle und Größe eines Samples extrahiert werden. Dies wird im weiteren Verlauf für die Aufteilung der Samples auf die CUDA-Kerne und Bestimmung der Frequenzen verwendet. Die Sampledaten werden mithilfe des „StreamWriter“-Subprojekts über eine TCP-Schnittstelle verfügbar gemacht. Diese TCP-Schnittstelle wird von einem weiteren Teilprojekt AudioTracer benutzt, um die Daten weiter zu verarbeiten. Das Modul **AudioTracer** liegt in zweifacher Ausführung in den Programmiersprachen C und Python vor. Beide Varianten liefern dieselbe Ausgabe und können somit auf der Ebene von Performance und Semantik verglichen werden. Die Aufgabe der Implementierung ist, die über die TCP-Schnittstelle empfangenen Audio Sampledaten in eine für CUDA passende Form zu bringen und diese mithilfe von einigen API-Anweisungen und Verteilungsentscheidungen auf die CUDA-Kerne der GPU zu partitionieren. Jeder dieser CUDA-Kerne führt dann eine Fouriertransformation auf den jeweils zugesicherten Teilbereich durch. Da die Audiodaten sehr viel mehr Samples beinhalten, als CUDA-Kerne vorhanden sind, muss vorher entschieden werden, wie viele Samples auf einmal bearbeitet werden können.

5.2 StreamWriter(C/C++)

Die Aufgabe des Teilprojekt StreamWriter ist es, Audiodateien einzulesen und diese Daten über eine Schnittstelle für andere Teilprojekte verfügbar zu machen. Um die Header-Informationen zu extrahieren, wird die Datei im Binärmodus geöffnet. Bestimmte Bereiche des Headers sind mit verschiedenen Größen versehen. Die Verwendung eines „typedef struct“ ermöglicht es, die Größen der Variablen vordefinieren zu können. Im weiteren Verlauf werden alle Headerinformationen durch diese Technik herausgefiltert. Die wichtigen bzw. in diesem Projekt verwendeten Headerinformationen sind Anzahl der Audiokanäle (NumChannels), Sample Rate (in der Wiedergabe abgespielte Audio-samples pro Sekunde), Bitanzahl pro Sample (BitsPerSample), sowie die eigentlichen Samples. Die Daten liegen in Hexadezimalform im Little-Endian Format vor. Diese Hexadezimalen Werte werden im Integer Format gespeichert. Der hier beschriebene Teil des Programms „StreamWriter“ ist in der Datei WAV.c bzw. WAV.h implementiert. Die Implementierung wird in allen weiteren Varianten verwendet, um eine WAV-Datei einzulesen und als Integer-Array speichern zu können. Analog zur Zielsetzung, werden die gespeicherten Audiodaten mithilfe einer Schnittstelle für das AudioTracer Programm zur Verfügung gestellt. Im Laufe der Bearbeitung dieser Arbeit sind dabei zwei Generationen von Schnittstellen entstanden, auf die im Folgenden Bezug genommen wird. Die grundsätzliche Idee war es, die Sampledaten dem AudioTracer als kontinuierlichen Datenstrom zur Verfügung zu stellen. Die erste Version erzeugt eine Datei, in der die Daten der Samples als Integer-Werte geschrieben wurden. Diese wurden an dieser Stelle im AudioTracer wieder stückweise abhängig eines definierten Thresholds eingelesen. Die zweite Generation dieser Schnittstelle sieht vor, die Daten über das Netzwerkprotokoll TCP zu versenden. Das AudioTracer-Modul greift diese Verbindung auf der anderen Seite auf und liest die Daten in vordefinierten Größen ein.

Datei-Variante In dieser Generation der Datenschnittstelle werden die Audiodaten über eine „Vermittlerdatei“ an das AudioTracer-Projekt weitergeleitet. Die zuvor in Integer umgewandelten Werte werden in hexadezimaler Form in die Datei „out.dat“ geschrieben. Jeweils 1000 Werte je Audiokanal werden pro Iteration der Routine in dieser Datei hinzugefügt. An dieser Stelle entsteht folgendes Problem: Der Zugriff auf eine Datei erfolgt sequentiell, d.h., es ist nur jeweils ein Zugriff zur gleichen Zeit auf eine Datei systembedingt gestattet. Das AudioTracer-Modul verlangt lesenden Zugriff auf die zuvor geschriebenen Daten. Im StreamWriter-Modul muss die Datei somit zwangsweise geschlossen werden, damit ein weiterer Zugriff seitens des AudioTracers stattfinden kann. Hier kann es unter Umständen zu einem Starvation-Problem kommen, wodurch das AudioTracer Modul keinen lesenden Zugriff auf die Datei erlangen kann. Im StreamWriter muss daher eine „Zwangspause“ in der Schleifeniteration implementiert werden. Um beiden Modulen einen Zugriff zu ermöglichen, wird diese Pause durch `Sleep(rand() % 50)` realisiert. Eine Randomisierung löst das Problem, dass auch bei festen Wartezeiten eine solche Starvation-Situation entstehen kann, wenn z.B. die Ausführungszeiten des AudioTracer-Moduls immer leicht über der vordefinierten Pause liegen.

TCP-Variante Diese Generation der Schnittstelle wird anstelle einer Datei, die als Vermittler dient, durch das TCP-Netzwerkprotokoll realisiert. Diese Variante des StreamWriter-Moduls wartet auf eine erfolgreiche Verbindung auf der anderen Seite der TCP-Verbindung. Sobald diese Verbindung besteht, werden die Integer Daten in einen String von Hexadezimalwerten umgewandelt. Die Umwandlung in Hexadezimal erfolgt, um eine gleichbleibende Größe der TCP-Pakete zu gewährleisten. Bei einem 16Bit-PCM WAV-Format ist ein Sample in einem Kanal jeweils zwei Byte groß, sodass vier Byte pro Zeitpunkt an Speicher benötigt werden. Die hexadezimalen Werte lassen sich leicht durch die Verwendung eines `stringstream`-Objekts in ein String Objekt umwandeln, welches für das Senden von Daten benötigt wird. Wie im vorherigen Abschnitt beschrieben, werden pro Iteration 1000 Samples pro Audiokanal gesendet und somit beträgt die Größe der versendeten Strings bei der TCP-Variante 10000 Bytes, da die Audiokanäle durch Leerzeichen und die einzelnen Sample-Gruppen durch ein `\n` Steuerzeichen abgegrenzt sind. Eine „Zwangspause“ wie in der ersten Generation ist hier durch die Verwendung des TCP-Protokolls nicht nötig, da eine direkte Pipeline zwischen AudioTracer- und StreamWriter-Modul besteht.

5.3 Code Architektur

In diesem Kapitel werden die Architektur des Projekts sowie prägnante Codestellen, die unter anderem Meilensteine zur Lösung des Ausgangsproblems darstellen, präsentiert. Der grundsätzliche Aufbau kann der nachfolgenden Abbildung 3 entnommen werden. Die Hauptfunktionen der einzelnen vorgestellten Module werden mithilfe von Sequenzdiagrammen in den jeweiligen Sektionen visualisiert, um einen leichten Überblick gewährleisten zu können.

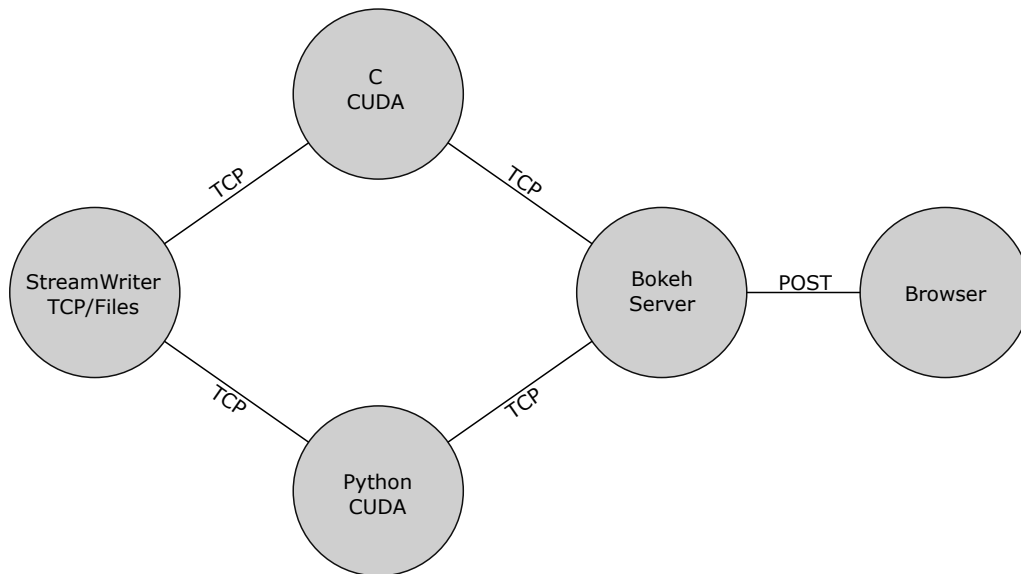


Abbildung 3: Projektüberblick

Das StreamWriter-Modul kommuniziert mit jeweils dem C++ CUDA- bzw. dem Python CUDA-Modul mithilfe des TCP-Protokolls. Diese verarbeiten die darüber empfangenen Daten und senden sie über eine weitere TCP-Verbindung zu einem Modul **Bokeh Server**. Dies ist ein Webserver mit vorgefertigten Techniken zur visuellen Aufbereitung von Daten. Die empfangenen Daten werden per AJAX-POST Anfrage an den Browser weitergeleitet und mit Javascript zur Laufzeit visualisiert. Die Module beschreiben jeweils eigenständige Programme, die über TCP-Verbindungen kommunizieren. Der generelle Ablauf geschieht nach Abbildung 3 von links, nach rechts. Im Projektverzeichnis des Github Repositories [3] befindet sich ein Batch-Skript zur Automatisierung dieses Ablaufs, das den Startvorgang der einzelnen Module in der gewünschten Reihenfolge gewährleistet und für einen reibungslosen Ablauf sorgt. Zunächst wird das Modul StreamWriter gestartet. Danach wird dem Benutzer die Entscheidung über die CUDA-Variante durch eine Benutzereingabe überlassen. Das Python- bzw. C++-Modul startet durch die Eingabe '1' bzw. '2'. Eingehende TCP-Pakete des StreamWriters beschreiben die Audiodaten und werden je nach ausgewähltem Modul bzw. Sprache anders verarbeitet. Die Ausgabe an den Bokeh-Server liegt jedoch bei beiden Modulen in gleicher Form vor. Aus den eingehenden periodischen Signalen entstehen somit über die verschiedenen

Verarbeitungsschritte und parallelen Abläufe klare Abbildungen, die in einer zugrundeliegenden Audiodatei anteilige harmonische Frequenzwerte visualisiert.

Im Folgenden werden nähere Einzelheiten der einzelnen Module thematisiert.

5.3.1 StreamWriter

Das Submodul `StreamWriter` dient als Schnittstelle zur Aufbereitung der Audiodaten aus den WAV-Dateien und der Sammlung dieser Daten bis zu einem bestimmten Schwellwert, sowie anschließender Weiterleitung über eine TCP Verbindung zum C++ bzw. Python CUDA-Modul. Die Main-Routine implementiert ein `fileConfig`-Objekt, in dem die wichtigsten Angaben gebündelt werden. Die Variable `format` kann die beiden Werte `DataFormat::WAV` und `DataFormat::RAW` annehmen. Erstere Angabe interpretiert die Eingabedatei als vollwertige WAV-Datei. Dabei müssen die in Kapitel 3.1 beschriebenen Headerinformationen ausgewertet werden. `DataFormat::RAW` betrachtet die Datei als Rohformat. Hierfür müssen jedoch wichtige Angaben, die normalerweise im Header gespeichert sind, angegeben werden. Die wichtigsten Angaben sind:

- Anzahl der Audiokanäle (`int channels`)
- Größe der einzelnen Samples in Byte (`int BytesPerSample`)
- Sample Rate (`int sampleRate`)
- Schwellwert der gleichzeitig verarbeiteten Audiosamples (`int threshhold`)

Außerdem wird hier der Pfad zur Eingabedatei in der Variablen `std::string inFile` spezifiziert. Beide Fälle der Eingabeformate durchlaufen eine ähnliche Routine und werden mithilfe eines Switch-Case-Konstrukts differenziert. Der gravierende Unterschied zwischen beiden ist, dass für das Format `DataFormat::RAW` keine Headerinformationen zur Verfügung stehen und diese somit nicht aus der Datei extrahiert werden müssen. Daher liegt der Fokus im Folgenden auf dem Fall einer Angabe des Formats `DataFormat::WAV`.

```
17 struct fileConfig
18 {
19     DataFormat format;
20     int bytesPerSample;
21     int sampleRate;
22     std::string inFile;
23     int channels;
24     int threshhold;
25 };
```

Abbildung 4: `fileConfig` struct

Der DataFormat::WAV-Fall des Switch-Case ist in Abbildung 8 demonstriert. Der Ablauf beginnt in Zeile 57 mit der Erstellung eines WAVLoader Pointers mit `WAVLoader* = new_WAVLoader((char*)file_in.c_str());`. An dieser Stelle wird die Eingabedatei dem WAVLoader Objekt übergeben, welches Headerinformationen und die eigentlichen Audiodaten extrahiert und in einem - abhängig von der Anzahl der Audiokanäle - zweidimensionalen Array speichert. Ein Meilenstein bei der Erstellung des WAVLoader-Objekts war die Erkenntnis, dass das Lesen von Dateien, welche im Binärformat vorliegen, auch im Binärmodus geöffnet werden müssen. Die Funktion `fopen_s` akzeptiert als dritten Parameter diese Art von Zugriffsmodus. Die Angabe von `'rb'` garantiert dabei lesenden Zugriff auf die Datei im Binärmodus. Während dies bei herkömmlichen Textdateien irrelevant ist, werden durch die Nichtangabe des Schalters „b“ die Steuerzeichen „\n“ und „\r“ als Daten übersetzt.

```

23     fopen_s(&self->fp, file, "rb");
24     if (!self->fp) {
25         printf("ERROR while opening file\n");
26         exit(1);

```

Abbildung 5: Eingabe Datei im Binärmodus lesen

Dies würde bei WAV-Dateien dazu führen, dass die Hexadezimalwerte z.B. „0x0A - 10“ und „0x0D - 13“ als Steuerzeichen interpretiert werden, sodass falsche Eingabewerte übersetzt würden. In Grafik 6 ist zu erkennen, wie schnell kleine Fehler zu falschen Ergebnissen führen können. Aus diesem Grund wird hier die Gegenüberstellung zwischen dem falschen (oben) und dem richtigen (unten) Eingabesignal präsentiert.

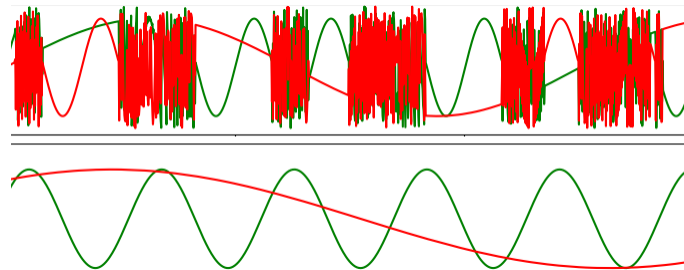


Abbildung 6: Vergleich `fopen_s` mit Modi „r“ (oben) und „rb“ (unten)

Der untere Graph zeigt auch unter anderem die erstellte Testdatei, in der zwei harmonische Sinuswellen (300hz und 40hz) auf jeweils dem linken bzw. rechten Audiokanal überlagert sind. Im weiteren Verlauf der Instanziierung des WAVLoader-Objekts werden hauptsächlich Daten der Ausgangsdatei mithilfe der Funktion `fread()` ausgelesen und in Membervariablen gespeichert. Das zuvor genannte zweidimensionale Array, welches die Sampledaten speichert, wird wie folgt realisiert:

```

1  int **channelArr = (int**) malloc(sizeof(int*) * self->fmt.NumChannels);
2
3  for (int i = 0; i < self->fmt.NumChannels; i++)
4  channelArr[i] = (int *)malloc(sizeof(int) * (self->data.SampleNumber) + 1);

```

Die Daten der Kanäle werden auf Integer-Pointer abgebildet, die wiederum von Integer Pointern referenziert sind. Deren Größe skaliert mit der Anzahl der Samples bzw. Anzahl der Audiokanäle. Je nach Eingabegrößen muss der entsprechende Speicherplatz hier angepasst werden.

Abbildung 7 zeigt die Speicherung der Daten in diesen beschriebenen Arrays. Anzumerken ist, dass die einzelnen Samples in einen Buffer der entsprechenden Größe eines Samples gelesen werden. Möglicherweise muss für die anschließende Zuweisung im Pointer eine Anpassung der Bytereihenfolge (Endianness) stattfinden. Dazu wird der Inhalt des Buffers durch `memcpy(&num, buffer, 4);` in einen 4 Byte Integer umgewandelt und in der entsprechenden Bytereihenfolge gespeichert.

Die Sample-Daten der WAV-Datei sind an dieser Stelle in den Speicher geladen und durch Pointer im WAVLoader Objekt zugänglich. Der nächste Schritt in Zeile 58 8 ist die Erstellung eines `Sender()`-Objekts. Der Konstruktor ruft die Membermethode `textttInitialize(std::string host, std::string port)` auf, die einen Socket auf Host „localhost“ mit dem Port 1337 bindet und auf eingehende Verbindungen wartet. Dafür wurde die Windows API verwendet, demnach ist das Programm nur auf dieser Plattform lauffähig. Für die Kompatibilität mit Linux-Distributionen müssten an dieser Stelle Änderungen implementiert werden. Die implementierte Funktion `int Sender::Send(std::string msg)` sendet den angegebenen String über den TCP-Socket an die entsprechenden angedockten Endpunkte.

```

91     if (self->fmt.AudioFormat == 1)
92     {
93         for (long i = 1; i <= self->data.SampleNumber; i++)
94         {
95             read = fread(data_buffer, self->data.SizePerSample, 1, self->fp);
96             if (read)
97             {
98                 int channels = 0;
99                 int channelData = 0;
100                 int offset = 0;
101
102                 for (; channels < self->fmt.NumChannels; channels++)
103                 {
104                     channelData = self->functions->convertEndianWithOffset(
105                         data_buffer, offset, bytesPerChannel);
106                     channelArr[channels][i-1] = channelData;
107                     offset += bytesPerChannel;
108                 }
109             }
110             else
111             {
112                 // redo iteration - should not normally happen
113                 printf("ERROR READING FILE");
114                 i--;
115             }
116         }
117     }

```

Abbildung 7: Hauptiteration: Extraktion der Sample Daten

```

54     case DataFormat::WAV:
55     {
56         printf("File: %s opened format %u\n", file_in.c_str(), format);
57         WAVLoader* wl = new_WAVLoader((char*)file_in.c_str());
58         Sender* sender = new Sender();
59
60         int index = 0;
61         std::stringstream ss;
62         while (true)
63         {
64             int upper = index + cfg.threshold;
65             ss.str("");
66             int bytes = 0;
67
68
69             for (;index < upper && index < wl->data.SampleNumber ; index++)
70             {
71                 for (int ch = 0; ch < wl->fmt.NumChannels; ch++)
72                 {
73                     uint16_t sample = unsigned(wl->data.channelArr[ch][index]);
74                     ss << std::setfill('0') << std::setw(4) << std::hex << static_cast<
                        uint32_t>(sample);
75                     if (ch < wl->fmt.NumChannels - 1)
76                         ss << " ";
77                 }
78                 ss << "\n";
79             }
80
81             sender->Send(ss.str());
82
83             if (index >= wl->data.SampleNumber)
84                 break;
85         }
86         sender->Shutdown();
87         delete wl;
88         delete sender;
89         break;
90     }
91 }
92 free(channelArr);

```

Abbildung 8: StreamWriter Main Loop

Die Zeilen 62 bis 85 in Abbildung 8 implementieren eine Schleifenoperation, in der die zuvor gespeicherten Sample-Daten in einem `stringstream` zusammengesetzt werden. Die Samples sind in Hexadezimalform im String gespeichert. Der Vorteil liegt darin, dass sich die Anzahl der Zeichen für die repräsentierten Dezimalzahlen nicht vergrößert. Die Samplegröße bestimmt die Bytengröße und somit die Länge der Hexadezimal-Strings. Um eine gleichbleibende String-Struktur beizubehalten, werden die Hexwerte wenn nötig auf die passende Länge erweitert und mit voranstehenden Nullen aufgefüllt.

```
1 ss << std::setfill('0') << std::setw(4) << std::hex << static_cast<uint32_t>(
    sample);
```

Anschließend wird der mit dem `stringstream` Objekt konstruierte String mit der Funktion `Send()` dem Sender Objekt über die TCP-Schnittstelle verfügbar gemacht.

An dieser Stelle ist die Aufgabe des StreamWriter-Moduls abgeschlossen. Die präsentierte Endlosschleife wird so lange durchgeführt, bis alle Sampledaten des WAVLoader-Objekts erfolgreich auf der Gegenseite empfangen wurden. Zuletzt erfolgen diverse Aufräumarbeiten bzw. Memory Cleanup.

```
1 sender->Send(ss.str());
```

Abbildung 9 veranschaulicht diesen Ablauf als Sequenzdiagramm.

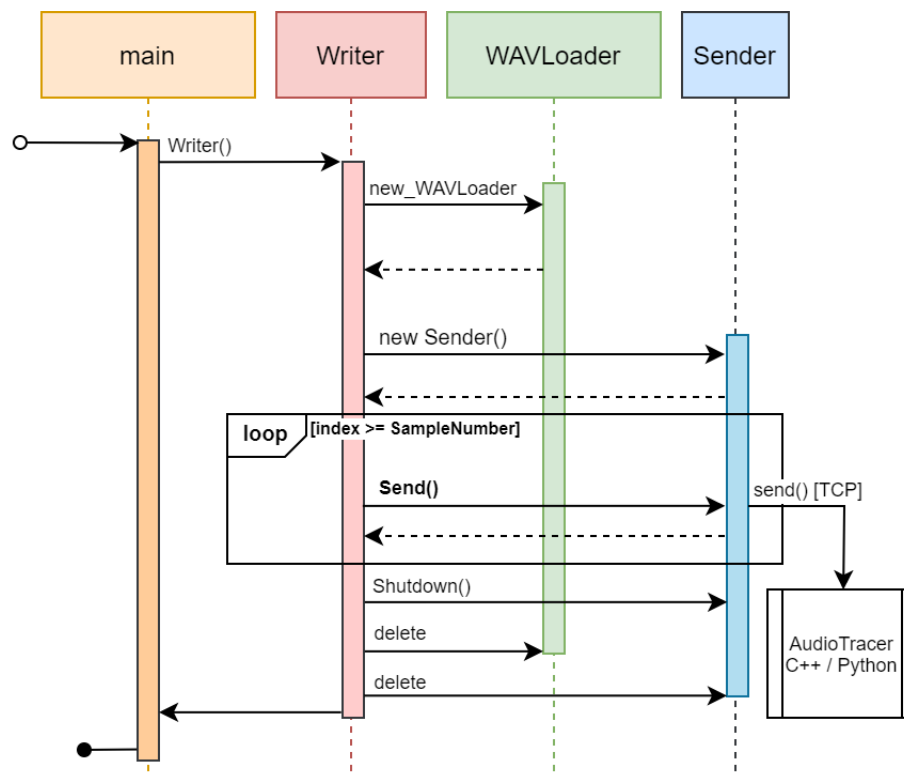


Abbildung 9: Sequenzdiagramm StreamWriter

5.3.2 AudioTracer

Die nächsten zwei Sektionen umschließen den Hauptteil des Programmierungsaufwands dieser Arbeit: Die Implementierung der Module AudioTracer in C- und Python- Variante. Die beiden Module erfüllen dieselbe Aufgabe, arbeiten jedoch unabhängig voneinander und werden nicht parallel ausgeführt. Die zu verarbeitenden Daten werden vom StreamWriter-Modul über bereits angesprochene TCP-Schnittstellen empfangen. Dafür werden analog zum StreamWriter Modul Socket Implementierungen vorgenommen, die die Audiosamples empfangen. Die Sockets knüpfen dabei aktiv an die horchende Gegenseite des StreamWriters an. Als Vorgriff ist anzumerken, dass hier bereits die Verbindung zum Bokeh-Visualisierungsmodul initialisiert wird. Dieser Teil wird später in der Sektion Bokeh Server zur Livevisualisierung thematisiert. Im Folgenden unterscheiden sich die Implementierungen der AudioTracer Module und werden daher getrennt genauer beschrieben.

AudioTracer (C) Die C Version des AudioTracers implementiert die Methode `initSocket()`. Aufgabe dieser Methode ist es, eine Verbindung zu einer Host Adresse mit gegebenem Port herzustellen und aufrechtzuerhalten. Der grundsätzliche Ablauf zur Erstellung und Verbindung eines Sockets mit der entsprechenden Gegenstelle ist in der Windows API folgendermaßen zu erreichen:

```
1  bool initSocket(SOCKET& s, char* host, int port)
2  {
3      struct sockaddr_in server;
4      WSADATA wsa;
5      WSASStartup(MAKEWORD(2, 2), &wsa);
6      s = socket(AF_INET, SOCK_STREAM, 0);
7      server.sin_addr.s_addr = inet_addr(host);
8      server.sin_family = AF_INET;
9      server.sin_port = htons(port);
10     connect(s, (struct sockaddr*)&server, sizeof(server));
11     return true;
12 }
```

Abbildung 10: Socketinitialisierung

Die Ausgaben und Statusabfragen auf Konsistenz der erzeugten Objekte bzw. Rückgabewerte der Methoden werden in diesem Codestück zur Vereinfachung nicht aufgeführt, sind aber in Abbildung 10 vollständig aufgelistet. Im struct `socketaddr_in` werden alle Informationen der Verbindung festgehalten, welche in den Zeilen 7 bis 9 festgelegt werden. Die Host-Adresse liegt als char-Pointer vor und muss mithilfe der Funktion `inet_addr(char*)` in den Typ `in_addr` umgewandelt werden (Zeile 7). Selbiges gilt für die Portnummer, welche als Integer vorliegt und in den Typ `USHORT` mit `htons()` umgewandelt wird (Zeile 9). Die Variable `sin_family` hat den Typ `ADDRESS_FAMILY`

und kann die Werte AF_UNSPEC, AF_INET und AF_INET6 annehmen. Hier wird der Wert AF_INET verwendet. Dadurch wird die Verbindung über das IPv4 Protokoll aufgebaut. AF_INET6 wäre für eine IPv6 Verbindung anzugeben. Die Winsock-DLL wird über folgende Funktion initialisiert:

```
1  int WSASStartup(
2  WORD      wVersionRequired,
3  LPWSADATA lpWSADATA
4  );
```

MAKEDWORD(2, 2) in Zeile 5 gibt dabei die zu verwendende Version der Winsock-DLL an. In diesem Fall wird Version 2.2 vorausgesetzt. Der Socket wird in Zeile 6 konstruiert. Die Methode socket akzeptiert als ersten Parameter die verwendete ADDRESS_FAMILY. Auch hier wird AF_INET und somit IPv4 verwendet. Der zweite Parameter gibt den Typ des Sockets an. Für den Verwendungszweck ist nur der Wert SOCK_STREAM sinnvoll. Die offizielle Dokumentation [13] gibt dazu folgende Beschreibung her: „A socket type that provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. This socket type uses the Transmission Control Protocol (TCP) for the Internet address family (AF_INET or AF_INET6).“ TCP ist offensichtlich sinnvoll, um die erfolgreiche Übertragung der Daten zu gewährleisten. Der dritte Parameter ist das Protokoll des Sockets selbst. Durch die Angabe des Wertes 0 wird dieses vom Service Provider, in diesem Fall StreamWriter Socket, festgelegt.

```
1  SOCKET WSAAPI socket(int af, int type, int protocol);
```

Zuletzt wird die Verbindung durch den Aufruf der folgenden Methode aufgebaut:

```
1  int WSAAPI connect(SOCKET s, const sockaddr *name, int namelen);
```

An diesem Punkt akzeptiert der StreamWriter-Socket die Verbindung und startet somit den in Sektion StreamWriter beschriebenen Ablauf. Die nächsten Schritte der Main Methode des AudioTracer(C)-Moduls implementieren eine Schleifenoperation, die für das Empfangen, sowie die eigentliche Verarbeitung der Daten zuständig ist. Zunächst müssen dafür jedoch einige Vorbereitungen getroffen werden. Analog zum StreamWriter-Modul werden die Sampledaten in einem Array gespeichert. Dieses wird im Voraus mit entsprechendem Speicherplatz bis zu einem Schwellwert (THRESHOLD) initialisiert. Die Zeilen 1 bis 3 realisieren diesen Schritt in folgendem Codeschnipsel.

```
1  int** sampleData = (int**)malloc(sizeof(int*) * channelNumber);
2  for (int i = 0; i < channelNumber; i++)
3      sampleData[i] = (int*)malloc(sizeof(int) * THRESHOLD);
4
5  int arr_pos = 0;
6  long start = 0;
7  int rcvSize = (channelNumber * sampleSize + channelNumber - 1 + 1) * THRESHOLD;
8  char* rcvBuf = (char*)malloc(sizeof(char) * channelNumber * THRESHOLD);
9  char* startptr, *endptr;
```

Außerdem werden für die darauffolgende Schleifenoperation wichtige Variablen angelegt, die zur Positionsbestimmung (Z. 5f.) und Extraktion der Daten aus dem TCP-Stream (Z. 8f.) zuständig sind. Der innere Ablauf der `while(true)`-Anweisung lässt sich in zwei Bereiche aufteilen. Der hintere Teil der Schleife ist in Abbildung 11 demonstriert. Die Zeilen 301 bis 305 implementieren das Empfangen der Daten, die zuvor vom StreamWriter-Modul über die TCP-Schnittstelle gesendet wurden. Da im Regelfall ein kontinuierlicher Datenstrom beim Empfangen der Daten erwartet wird, werden so lange Daten von der Gegenseite angenommen, bis ein vom Schwellwert abhängiger Byte-Wert empfangen wurde. Die Variable `rcvSize` gibt die zu erwartende Bytegröße nach der Formel im obigen Codefragment (Z. 7) an. Für jede Sample-Gruppe wird die Anzahl der Kanäle mit der Samplegröße multipliziert. Die Kanäle werden mithilfe von Leerzeichen getrennt und von neuen Sample-Gruppen durch ein Newline-Steuerzeichen getrennt. Multipliziert mit der maximalen Anzahl der gleichzeitig verarbeiteten Sample-Gruppen ergibt dies die zu erwartende Größe der Daten, die über den TCP-Stream empfangen werden in Bytes an. Bei einem Schwellwert von 5000 Samples á vier Byte auf zwei Kanälen würden nach dieser Vorgehensweise 50000 Bytes empfangen werden.

```

301     while (bytes < rcvSize)
302     {
303         int rcvBytes = recv(s_rcv, rcvBuf + bytes, rcvSize - bytes, 0);
304         bytes += rcvBytes;
305     }
307     char delim[] = "\n";
309     unsigned count = 0;
310     startptr = endptr = (char*)rcvBuf;
313     char* channelStr = (char*)malloc(sizeof(char) * (channelNumber * sampleSize
        + (sampleSize-1) + 1));
314     while ((endptr = strchr(startptr, '\n')))
315     {
316         sprintf(channelStr, "%.*s", (int)(endptr - startptr + 1), startptr);
317         char* perChannel = strtok(channelStr, " ");
318         for (int c = 0; c < channelNumber; c++)
319         {
320             int channelData = (int)strtol(perChannel, NULL, 16);
321             perChannel = strtok(NULL, " ");
322             sampleData[c][arr_pos] = channelData;
323         }
324         startptr = endptr + 1;
325         arr_pos++;
326     }

```

Abbildung 11: Schleifenoperation: Hinterer Teil

Der Bereich von Zeile 307 bis 326 realisiert einen String Tokenizer, um aus dem empfangenen String die eigentlichen Daten zu extrahieren. Dabei werden wieder einzelne Sample-Gruppen mit einem Newline-Steuerzeichen und einzelne Kanaldaten mit einem Leerzeichen separiert und mit der Methode `strtol` in Integer-Werte konvertiert. Dafür zeigen zunächst zwei Pointer (`startptr` und `endptr`) auf den Anfang des empfangenen Charpointers `rcvBuf`. Danach wird das „\n“ Steuerzeichen lokalisiert und der Pointer `endptr` auf dessen Position ausgerichtet. Die Zeichen des Strings zwischen `endptr` und `startptr` werden darauf folgend in ein neu erzeugtes String-Objekt `char* perChannel` kopiert. Die Variable `perChannel` enthält nun wiederum die String-Repräsentation einer Sample-Gruppe, die mithilfe von `strtok()` in einzelne Kanaldaten aufgespalten und somit in Integer umgewandelt in das Array `sampleData` kopiert werden kann. Letztlich wird der Pointer `startptr` auf die Position von `endptr + 1` gesetzt, um von dort aus das nächste Auftreten eines „\n“ Steuerzeichens zu finden. Diese Vorgehensweise ist unter anderem ein Beispiel für einen nicht destruktiven String Tokenizer in C. Eine Variable `arr_pos` zählt die bereits übertragenden Sample-Gruppen. Sobald diese den Schwellwert `THRESHHOLD` übersteigt, wird der Teil des Programms erreicht, der eine hohe Relevanz in dieser Arbeit einnimmt: Die Implementierung der Parallelisierung einer Fouriertransformation auf den bis jetzt übermittelten und extrahierten Daten über die CUDA-Schnittstelle für Nvidia Grafikkarten.

Die Implementierung der parallelisierten Fouriertransformation gibt die folgende Methodensignatur vor:

```
1  cudaError_t cudaFourierTransform(cplx* x, cplx* complexOut, int channelNb, int
    size)
```

Die Parameter `x` und `complexOut` werden als Pointer vom Typ `cplx` erwartet. Dieser Typ ist eine Abstraktion der CUDA-eigenen Bibliothek für komplexe Zahlen in float-Genauigkeit. Die bereits empfangenen Daten liegen jedoch im Typ `Integer` vor. Aus Typkonsistenzgründen müssen diese zum Typ `cplx` konvertiert werden. Außerdem ist die Arbeit mit mehrdimensionalen Arrays - bzw. in diesem Fall Pointer-auf-Pointer Konstrukten - schwierig, sodass hier zusätzlich eine Modulation in ein eindimensionales Array der Daten durchgeführt wird. Die `cuComplex` Bibliothek stellt die Methode `make_cuComplex` zur Verfügung. Die Angabe von Real- und Imaginärteil einer komplexen Zahl wird dafür erwartet. Da die Ausgangsdaten im `Integer` Format vorliegen und somit keinen Imaginärteil besitzen, wird dieser mit 0 angegeben. Zeilen 276 bis 283 in Abbildung 12 realisieren diese beiden Schritte.

Die Ausgabe der Fouriertransformation erfolgt in einem neu angelegten Pointer vom Typ `cplx` mit entsprechend allozierten Speicherplatz, abhängig zur Anzahl der Audiokanäle und `THRESHHOLD` (Z. 273). Aufgrund des Stellenwerts, den die Methode `cudaFourierTranform` einnimmt, wird diese in der hierauf folgenden Sektion separat beschrieben. Wichtig für den hier thematisierten Teil der `AudioTracer` Implementierung ist, dass der Pointer `complexOut` auf die fouriertransformierten Rückgabewerte zeigt. In der Topologie der Implementierung nach Abbildung 3 ist die Beschreibung mit der Me-

```

271     if (arr_pos >= THRESHHOLD)
272     {
273         cplx* complexOut = (cplx*)malloc(sizeof(cplx) * channelNumber *
                THRESHHOLD);
274         cplx* helper = (cplx*)malloc(sizeof(cplx) * channelNumber * THRESHHOLD);
275
276         for (int i = 0; i < channelNumber; i++)
277         {
278             for (int j = 0; j < THRESHHOLD; j++)
279             {
280                 int sample = sampleData[i][j];
281                 helper[i * THRESHHOLD + j] = make_cuComplex(sampleData[i][j], 0)
                ;
282             }
283         }
284
285         cudaFourierTransform(helper, complexOut, channelNumber, THRESHHOLD);
286         sendBuf(s_vis, complexOut, THRESHHOLD, channelNumber);
287
288         start = arr_pos;
289         arr_pos = 0;
290
291         free(sampleData);
292         sampleData = (int**)malloc(sizeof(int*) * channelNumber);
293         for (int i = 0; i < channelNumber; i++)
294             sampleData[i] = (int*)malloc(sizeof(int) * THRESHHOLD);
295     }
296 }

```

Abbildung 12: Schleifenoperation: Vorderer Teil

thode sendBuf in Zeile 286 am Kantenübergang zwischen dem Modul AudioTracer(C) und Bokeh-Server angelangt. Die Daten werden über den anfangs erstellten Socket s_vis an den Server gesendet. Die grundsätzliche Vorgehensweise die Übermittlung der Daten in Form eines JSON-Strings. Da es sich bei dem Bokeh-Modul um einen Webserver handelt, ist eine solche Übertragung in gegenwärtiger JavaScript-Umgebung sehr sinnvoll. Die Daten müssen lediglich in die Form eines JSON-Arrays gebracht werden. Der nötige Schritt ist die Umschließung der kommaseparierten Daten mit „[“ bzw. „]“ und wird durch eine einfache Iteration in Verbindung mit der Funktion snprintf realisiert. Problematisch ist jedoch, dass die durch die Fouriertransformation entstandenen Ausgabedaten weiterhin im Format cplx* vorliegen. Komplexe Zahlen werden üblicherweise im kartesischen Koordinatensystem mit Real- und Imaginärteil angegeben. Mithilfe der Polarform lässt sich der Betrag einer komplexen Zahl errechnen:

Definition (6): Polarform von komplexen Zahlen

Sei z eine komplexe Zahl in der Form $z = a + b * i$.

Dann ist der absolute Betrag r definiert als: (6)

$$r = |z| = \sqrt{a^2 + b^2}$$

In der Implementierung wird dafür die Bezeichnung `ftMagnitude` verwendet:

```
1  float ftMagnitude = sqrtf(  
2      cuCreal(complx[c * threshold + samp])  
3      * cuCreal(complx[c * threshold + samp])  
4      + cuCimag(complx[c * threshold + samp])  
5      * cuCimag(complx[c * threshold + samp]));  
6  mag[c * threshold + samp] = ftMagnitude;
```

Die Variablen `c` und `samp` beschreiben im Kontext einer Iteration der Sampledaten den aktuell betrachteten Kanal bzw. Sample aus diesem Kanal. Die Libraryfunktionen `cuCreal` und `cuCimag` lesen den Real- bzw. Imaginärteil analog zur Definition a bzw. b der komplexen Zahl aus. Der zusammengesetzte String wird über den vorhandenen Socket an den Bokeh Server gesendet.

Der Ablauf des AudioTracer-C Moduls wird in Abbildung 13 als Sequenzdiagramm dargestellt und gibt alle wichtigen Abläufe und Funktionen des Programms an.

Der folgende Part liefert genauere Einblicke in die Implementierung der Initialisierung der CUDA-Schnittstelle, Speicherverwaltung, sowie die Aufteilung der parallelisierten Fouriertransformation auf CUDA-Kernels und letztlich die Implementierung der Fouriertransformation selbst.

Einzelheiten der CUDA Implementierung Die vorherigen Kapitel dieser Arbeit beschreiben einen großen Vorbereitungsaufwand, um eine ursprüngliche Audiodatei im WAV-Format für eine Implementierung einer Fouriertransformation verfügbar zu machen. Dabei wird die Datei interpretiert und deren Daten extrahiert. Diese werden über eine TCP-Schnittstelle versendet und von der Gegenseite empfangen, in einem entsprechenden Format bereitgestellt und für den nächsten Schritt vorbereitet.

Abbildung 14 zeigt die wichtigsten Codestellen und Überlegungen für den Abschluss der Vorbereitungsarbeiten der parallelen Transformation. Die Voraussetzung ist CUDA-fähige Hardware und die entsprechende Entwicklungsumgebung mit mindestens eingebundenen Cuda Bibliotheken `cuda_runtime.h` und `device_launch_parameters.h`. Zeile 340 initialisiert das CUDA-fähige Gerät durch den Aufruf der Methode `cudaSetDevice(int device)`. Die Spezifikationen des Entwicklungssystems ist in Sektion 4.3 beschrieben. Für diesen Fall steht ein Gerät zur Verfügung, somit ist diese mit dem Parameter 0 zu referenzieren. Bei Multi-GPU-Systemen würde an dieser Stelle der gewünschte Index angegeben. Informationen über die installierten GPU-Geräte können mithilfe von `cudaGetDevice(int device)` abgefragt werden. Für den Fehlerfall der CUDA-Funktionen steht der Rückgabotyp `cudaError_t` bereit.

Grundsätzlich kann die Device-Umgebung nicht direkt auf den Host Adressraum zugreifen. Dafür müssen die verwendeten Daten auf den Device Adressraum und Rückgabewerte von diesem wieder auf den Adressbereich des Host-Systems übertragen werden. Die Variablen `cplx* dev_c` und `cplx* dev_out` werden für diese Zwecke in den Zeilen 347f. auf dem Speicherbereich der Grafikkarte alloziert (`cudaMalloc`). Der anzugebende Pointer zeigt nach Ausführung von `cudaMalloc` auf den allozierten Speicher auf der Device Umgebung. Der Speicher für die Ausgabe muss lediglich auf der Grafikkarte initialisiert werden. Die Sampledaten müssen allerdings mit `cudaMemcpy` auf das Gerät kopiert werden. Die Parameter sind ein Pointer auf den Speicherbereich der Grafikkarte und ein weiterer, in dem sich die Daten befinden, die übertragen werden sollen. Die Richtung des Kopiervorgang muss hier explizit angegeben werden. Die nötigen Angaben sind `cudaMemcpyHostToDevice` für das Kopieren in Device Richtung und `cudaMemcpyDeviceToHost` für die Rückrichtung.

Für die parallele Ausführung von CUDA-Kernels muss vorher ein entsprechendes Muster angelegt werden. Hierfür werden dreidimensionale Vektoren verwendet. Ab einer gewissen Größe der Eingabedaten, in diesem Anwendungsfall Anzahl der Audiosamples, wird ein Maximum an parallel ausführbaren Kernels erreicht. Aus diesem Grund können nicht alle Daten parallel verarbeitet werden. Die auszuführenden Threads sind in einem System von CUDA-Grids und -Blocks angegeben.

```

337     cplx* dev_c;
338     cplx* dev_out;

340     cudaStatus = cudaSetDevice(0);
341     if (cudaStatus != cudaSuccess) {
342         fprintf(stderr, "cudaSetDevice failed!");
343         goto Error;
344     }

347     cudaStatus = cudaMalloc((void**)&dev_out, sizeof(cplx) * channelNumber * size);
348     cudaStatus = cudaMalloc((void**)&dev_c, sizeof(cplx) * channelNumber * size);
349     cudaStatus = cudaMemcpy(dev_c, x, sizeof(cplx) * channelNumber * size,
        cudaMemcpyHostToDevice);

351     int maxThreadsPerBlock = 1024;
352     int threadCluster = (int)ceilf((float)size / maxThreadsPerBlock);
353     int actualThreadsPerBlock = size;
354
355     if (threadCluster)
356         actualThreadsPerBlock = (int)ceilf((float)size / threadCluster);
357
358     dim3 block_dim(actualThreadsPerBlock, 1, 1);
359     dim3 grid_dim(threadCluster, channelNumber, 1);

362     addKernel <<<grid_dim, block_dim >>> (dev_c, dev_out, channelNumber, size);

370     cudaStatus = cudaDeviceSynchronize();
371     if (cudaStatus != cudaSuccess) {
372         fprintf(stderr, "cudaDeviceSynchronize returned error code %d after
        launching addKernel!\n", cudaStatus);
373         goto Error;
374     }

378     cudaStatus = cudaMemcpy(complexOut, dev_out, sizeof(cplx) * size * channelNumber
        , cudaMemcpyDeviceToHost);
379     if (cudaStatus != cudaSuccess) {
380         fprintf(stderr, "cudaMemcpy failed!");
381         goto Error;
382     }

```

Abbildung 14: CUDA Vorbereitungen

Je nach Dimension dieser als Vektoren angegebenen Variablen ändert sich die Anzahl der Threads auf den CUDA-Kernels.

In Abbildung 15 ist die Topologie der GPU abgebildet. Ein CUDA-Grid (links) besteht aus den rechts abgebildeten Blöcken (Block-0 ... Block-(K-1)). In jedem dieser Blöcke wird eine definierte Anzahl an Threads ausgeführt, die hier in wellenförmigen Pfeilen

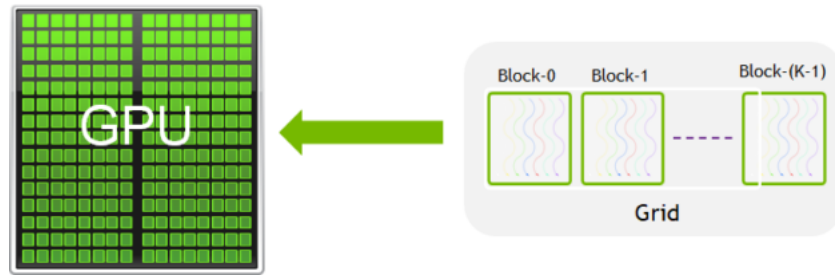


Abbildung 15: Topologie der CUDA Kernels [14]

dargestellt werden.

In Zeile 351 (Abbildung 14) wird die maximale Anzahl der Threads in einem Block (1024) für die in der Entwicklungsumgebung verwendeten GPU angegeben. Die weitere Berechnung der Grid- und Blockdimensionen basieren unter anderem auf dieser Angabe. Die Überlegung zur Aufteilung, welche in den Zeilen 352 bis 356 durchgeführt wird, geht dahin, dass die Grid-Dimensionen abhängig von der Anzahl der Audiokanäle sowie Anzahl der Samples wachsen. Die Variable `threadCluster` in Zeile 352 berechnet sich aus der Division der Eingangsgröße durch die maximale Anzahl der Threads in einem Block. Um Datenverlust zu verhindern, wird dieses Ergebnis auf den nächstgrößeren Integer Wert aufgerundet. Mithilfe der Variablen `channelNumber` und `threadCluster` können die Dimensionen des CUDA-Grids (Z. 358f) definiert werden. Dabei bestimmen `threadCluster` bzw. `channelNumber` die Größe in X- bzw. Y-Richtung. Die Länge der Z-Richtung des Vektors nimmt den Wert 1 an, da eine abweichende Angabe hier nicht notwendig ist. Die Größe der Blöcke im angegebenen Grid wird in Abhängigkeit der `threadCluster` berechnet und in der Variablen `actualThreadsPerBlock` gespeichert (Z. 353 u. 356). Die Größe wird mit einer Division der Anzahl von Eingabesignalen durch die zuvor berechneten `threadCluster` definiert. Die Notwendigkeit von mehrdimensionalen Blockvektoren wird hier nicht gesehen. Somit bestimmt die Variable `actualThreadsPerBlock` die Länge des Blockvektors in X-Richtung. Die Größen der Grid- und Blockdimensionen werden in Abbildung 16 visualisiert. Die Berechnung basiert auf folgenden Eingabegrößen:

- `maxThreadsPerBlock = 1024`
- `size = 5000`
- `channelNumber = 2`

Mithilfe der Abbildung wird die allgemeine Struktur der Kernausführung klarer.

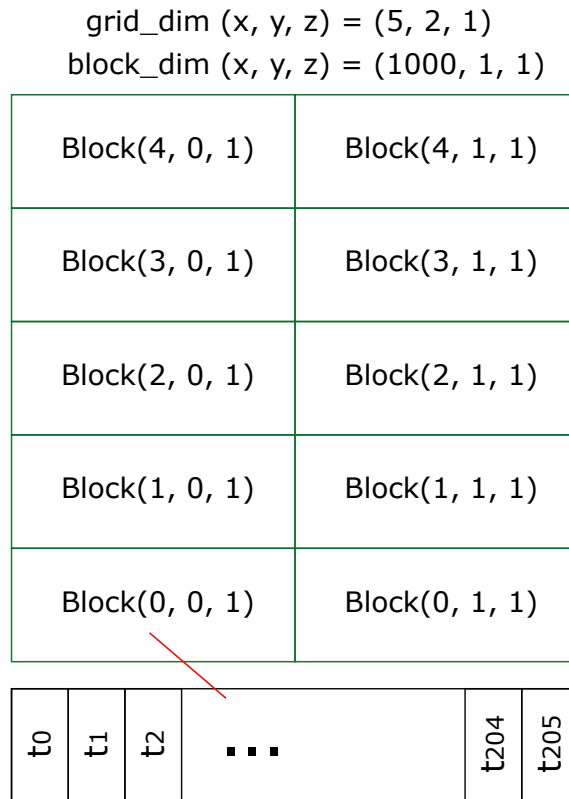


Abbildung 16: Gridbeispiel

Jeder ausgeführte Thread hat Zugriff auf die Variablen blockIdx, blockDim, gridDim und threadIdx. Damit lässt sich zu jeder Zeit identifizieren, an welcher Stelle im Grid sich der aktuell betrachtete Thread befindet. Diese Information wird für den Zugriff auf die Pointer der Sampledaten verwendet.

```

129 __global__ void addKernel(cplx* x, cplx* out, int channelNummber, long size)
130 {
132     _dft(x, out, size, blockIdx, blockDim, gridDim, threadIdx);
133 }

```

Abbildung 17: CUDA Kernel

Im Codebeispiel 17 ist die Implementierung der Kernelfunktion angegeben. In Zeile 129 ist zu erkennen, dass die Funktion als `__global__` deklariert ist. Diese Direktive ermöglicht das Aufrufen von Funktionen im Speicherbereich des Device Systems. In diesem Fall ruft das Host-System die Methode `addKernel` (Abbildung 14 Zeile 362) im Speicherbereich der Grafikkarte auf. Mithilfe des Operators „<<< >>>“ werden die zuvor berechneten Dimensionen von Grid und Blocks als Parameter überreicht. Analog dazu gibt es die Direktive `__device__`. Die so annotierten Funktionen können nur im

Adressbereich der GPU referenziert werden und sind über das Host System nicht erreichbar. Die globale Funktion `addKernel` ruft die Methode `_dft` auf, die das Herzstück der Kernaussführung bildet. Die Auflistung 18 liefert die Implementierung einer diskreten Fouriertransformation, die in jeweils einem CUDA-Kernel durchlaufen wird. In den Zeilen 36 bis 39 werden die notwendigen Index-Informationen berechnet. Da jeder Kernel Zugriff auf den Ausgabe Pointer hat, muss die Referenzierung der Pointerpositionen eindeutig sein. Abbildung 16 hilft dabei, diesen Vorgang zu veranschaulichen.

- `threadNumInBlock`: Gibt den Index des Threads im aktuellen Block an und wird durch $\text{threadIdx.x} + \text{blockDim.x} * \text{threadIdx.y}$ bestimmt. In der Skizze würde zum Beispiel Thread t_{200} vom Block(0, 1, 1) angenommen. `threadIdx.x` wäre hier 200 und `blockDim.x` = 1000. Die Threads sind eindimensional angegeben, deshalb betragen y und z Werte von `threadIdx` jeweils 1. Somit ergibt sich ein Threadindex innerhalb des Blocks bei gegebenem Beispiel von $200 + 1000 * 0 = 200$
- `side`: Gibt den Index in Abhängigkeit des Blockindexes in Y-Richtung an. Analog zur Abbildung beschreibt dies den Anfangsindex einer Seite mit gleicher Y-Koordinate im Blockindex und ist mit $\text{gridDim.x} * \text{blockIdx.y} * \text{blockDim.x}$ angegeben. Beispielsweise wäre der Index zum Anfang der „rechten“ Seite der Abbildung $5 * 1 * 1000 = 5000$
- `sideIndex`: Der Index innerhalb einer hier so genannten Seite lässt sich durch $\text{blockIdx.x} * \text{blockDim.x} + \text{threadNumInBlock}$. Für den im ersten Punkt genannten Thread lautet der Index $0 * 1000 + 200 = 200$.
- `globalIndex`: Gibt den globalen Index des Threads im Gridsystem an. Beide dafür notwendigen Parameter `side` und `sideIndex` sind dafür bereits gegeben und werden addiert: $5000 + 200 = 5200$

Die Variablen `sideIndex` und `globalIndex` werden für die Berechnung der Fouriertransformation sowie für die Positionsbestimmung im Ausgabearray verwendet. In (2) wurde bereits die Definition der diskreten Fouriertransformation vorgestellt. Mithilfe von (3) wird klar, dass der Exponent $e^{\frac{-ikl \cdot 2\pi}{N}}$ in Real- und Imaginärteil umgewandelt werden kann: $\cos(\frac{2\pi}{N}kl) - i \cdot \sin(\frac{2\pi}{N}kl)$. Diese Formel passt in sofern auf die Implementierung, dass die Exponenten im Typ `cplx` vorliegen müssen, um die weiteren Rechenschritte ausführen zu können. Zeile 44 führt diesen Schritt aus, wobei die eben genannten Indizes `sideIndex` und `globalIndex` verwendet werden. Nach der Formel wird dieser Exponent mit `Audiosample` multipliziert und das Ergebnis zur Ausgabe aufsummiert. Analog wird in Zeile 45 `cuCmulf` bzw. `cuCaddf` verwendet. Der Dämpfungsfaktor $1/N$ wird in der Implementierung nicht verrechnet, da dieser keine Auswirkung auf die zu observierenden Ergebnisse hat.

Das Hostsystem wartet durch den Methodenaufruf `cudaDeviceSynchronize()` im Programmablauf auf die Beendigung aller erstellten Threads. Sind diese erfolgreich zum Abschluss gekommen, muss das Ausgabearray `dev_out` im Speicherbereich der GPU wieder in das Host System kopiert werden. Dies wird realisiert durch einen erneuten

Aufruf von `cudaMemcpy` mit der Richtungsangabe `cudaMemcpyDeviceToHost`.

Bis hierher wurde ein erfolgreicher Durchlauf einer parallel ausgeführten Fouriertransformation beschrieben. Das Hostprogramm wartet nun auf Pakete, die zur TCP-Schnittstelle gesendet werden, um die Iteration erneut zu durchlaufen.

```
32 __device__ void _dft(cplx* buf, cplx* out, int N, dim3 blockIdx, dim3 blockDim,
    dim3 gridDim, dim3 threadIdx)
33 {
34     double PI = acosf(-1);
35
36     int threadNumInBlock = threadIdx.x + blockDim.x * threadIdx.y;
37     int side = gridDim.x * blockIdx.y * blockDim.x;
38     int sideIndex = blockIdx.x * blockDim.x + threadNumInBlock;
39     int globalIndex = side + sideIndex; //global thread number;
40
41     for (int j = 0; j < N; j++)
42     {
43         int index_j = side + j;
44         cplx exp = make_cuComplex(cos(((2 * PI) / N) * sideIndex * j), -1 * sin(((2
            * PI) / N) * sideIndex * j));
45         cplx res = cuCaddf(out[globalIndex], cuCmulf(buf[index_j], exp));
46         out[globalIndex] = res;
47     }
48 }
```

Abbildung 18: Kernelfunktion `_dft`

5.3.3 AudioTracer (Python)

Die Python Version des AudioTracer Projekts weist generell die gleiche Semantik wie das C Modul auf, hat jedoch einige Besonderheiten. Vor allem fällt die Kompaktheit des Python Moduls direkt auf. In ca. 100 Zeilen Code werden Daten empfangen, verarbeitet (fouriertransformiert) und zum Bokeh Modul weiter versendet. Dies liegt unter anderem an vielen hilfreichen Operatoren und vorgefertigten Methoden, die ein künstliches aufblähen des Codes verhindern. So wird z.B. in drei Zeilen Code (Abbildung 19) ein voll funktionsfähiges Socket Objekt erstellt, über das Daten gesendet bzw. empfangen werden können.

```
47 with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
48     s.connect((HOST, PORT))
49     s.setsockopt(socket.IPPROTO_TCP, socket.TCP_NODELAY, 1)
```

Abbildung 19: Erstellung eines Sockets in Python

Beim Umgang mit den Empfangenen Daten des StreamWriter-Moduls ersetzt die Funktion `split()` aufwändige Pointer-Arithmetik, wie sie im C-Modul verwendet wurde. Auch die Konvertierung der Hex-Strings wird in einem Einzeiler durchgeführt:

```
1 sample = int(perChannel[c], 16)
```

Gegebenenfalls müssen hier die verwendeten Bits bei größeren Datentypen der Eingangssignale angepasst werden. Auch das Senden der fouriertransformierten Daten über den Socket des Bokeh Server funktioniert mit geringerem Aufwand:

```
1 json_str = json.dumps(freq_magnitudes)
2 send_data(visSocket, (str(len(json_str))+'|').encode('utf-8'))
3 send_data(visSocket, json_str.encode('utf-8'))
```

Die bereitgestellte Methode `dumps` des `json` Packages ermöglicht es das Array der Ausgabedaten in einen JSON-String zu konvertieren. Dieses kann ohne weitere Stringmanipulationen über den Socket des Bokeh Webservers gesendet bzw. empfangen werden. Folgende Abbildung 20 zeigt eine Implementierung der Fast Fourier Transformation (FFT) mithilfe der `cupy`-Library und anschließende Umwandlung der komplexen Ausgabedaten in deren absoluten Beträge.

```

53         data_t_gpu = cp.array(sampleData)
54         data_o1_gpu = cp.fft.fft(data_t_gpu, axis=1)
55         cp.cuda.Device().synchronize()
56         freq_magnitudes = list()
57
58         for arr in data_o1_gpu:
59             for cplx in arr:
60                 magnitude = math.sqrt(np.real(cplx) * np.real(cplx) + np.
61                                     imag(cplx) * np.imag(cplx))
62                 freq_magnitudes.append(magnitude)

```

Abbildung 20: Ausführung CUDA FFT in Python

In Zeile 53 wird das Array der Sample Daten im Speicherbereich der GPU verfügbar gemacht. Dafür wird die Funktion `array()` des `cupy` Pakets verwendet. Der Rückgabewert ist ein Verweis auf das neu angelegte Array im Device Speicher. Zeile 54 implementiert die Durchführung der FFT basierend auf dem zuvor erstellten Array. Die Methode `fft` gibt die Daten direkt zurück. Diese müssen nicht manuell vom Device- zum Hostsystem übertragen werden.

Der weitere Ablauf des Python Moduls verläuft analog zum C-Äquivalent. Nach der erfolgreichen Abschluss der FFT und dem Warten auf Beendigung aller CUDA Threads, werden die Daten zum Bokeh Server gesendet. Anschließend werden neue Daten am TCP-Socket des StreamWriter-Moduls erwartet.

Das Python-AudioTracer-Moduls wird in Abbildung 21 erneut in Form eines Sequenzdiagramms angegeben. Dieses weist eine ähnliche Struktur zum C-Äquivalent auf, benutzt jedoch andere Bezeichnungen und externe Strukturen, wie z.B. der Methode `cufft` anstatt `addKernel` etc.

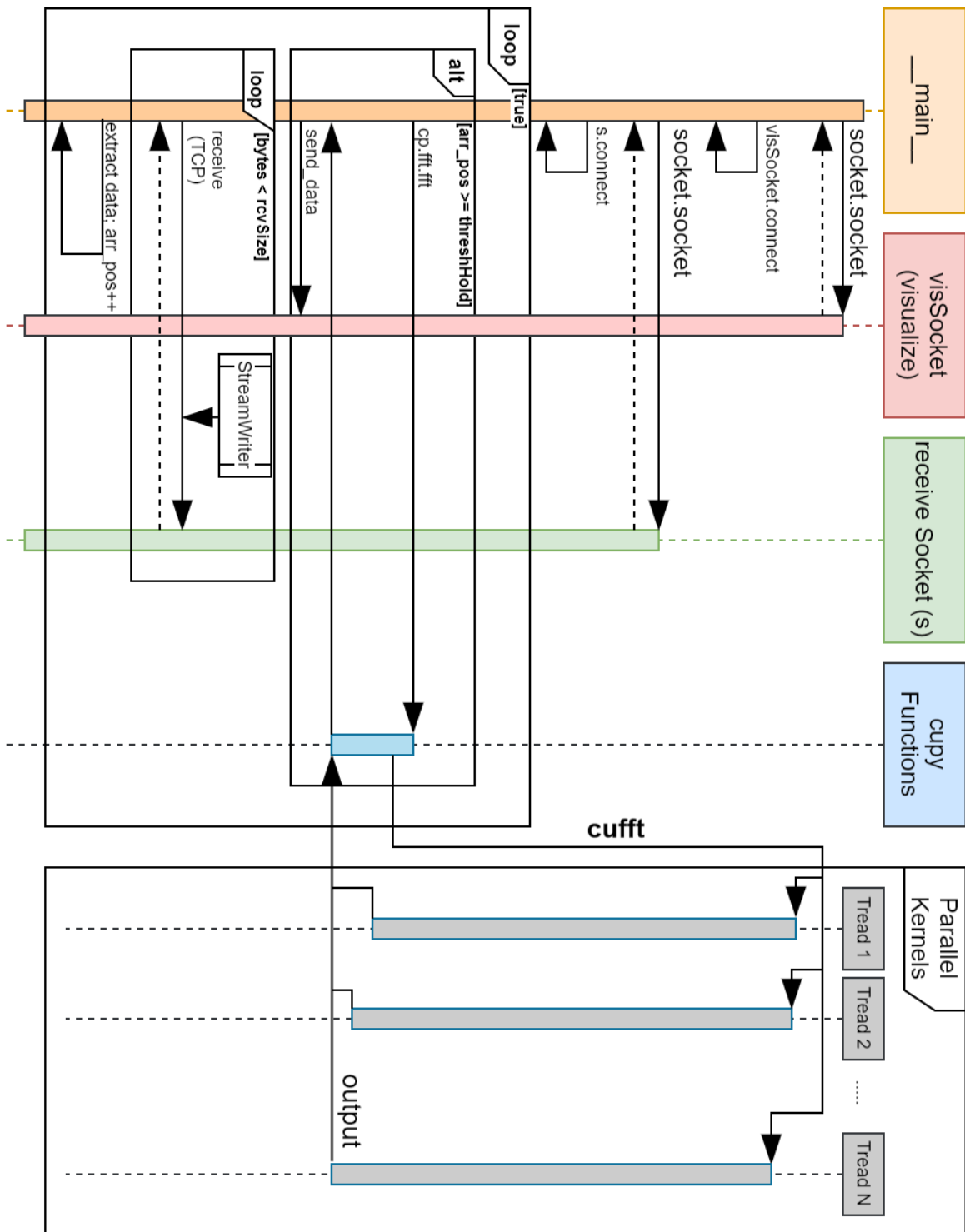


Abbildung 21: Sequenzdiagramm AudioTracer-Python

5.3.4 Bokeh Server zur Livevisualisierung

Das Bokeh Modul besteht aus einem Webserver, der individuelle Datenmengen an den Browser übertragen kann, der diese mithilfe von JavaScript in eine Grafik visualisiert. Bokeh erlaubt die Übertragung von Daten über POST-Requests an eine vorgegebene Daten-URL. Die Erstellung der nötigen Komponenten und Parameter wird in Abbildung 22 demonstriert.

```
77 port = 8086
78 source = AjaxDataSource(data_url=f'http://localhost:{port}/data',
79                          polling_interval=1000, adapter=adapter)
80
81 TOOLTIPS = [
82     ("index", "$index"),
83     ("(x,y)", "($x, $y)"),
84 ]
85
86 p = figure(plot_height=500, plot_width=1800, background_fill_color="lightgrey",
87            title="Fourier Frequencies", tooltips=TOOLTIPS)
88 p.x_range = Range1d(-25000, 25000)
89 p.y_range = Range1d(0, 1000000000)
90 p.circle_dot('x', 'y', source=source)
```

Abbildung 22: Erstellung eines Plots mit Ajax Datenübertragung

Die Datenquelle ist durch die Methode `AjaxDataSource` vorgegeben (Z. 78). Die drei angegebenen Parameter geben folgende Informationen an:

- `data_url`: Angabe der URL, auf der die Daten empfangen werden
- `polling_interval`: Angabe des Polling Intervalls der Daten auf gegebener URL in Millisekunden
- `adapter`: Ein individuell erstellter JavaScript-Adapter, der angibt wie die empfangenen Daten an die Visualisierung weitergegeben werden. Dazu mehr im Kontext von Abbildung 22

Es besteht die Möglichkeit, die Datenpunkten mit individuellen Tooltips zu verknüpfen. In Zeilen 81 bis 84 wird ein Tooltip erstellt, der die eindeutige ID, sowie die Lage im Koordinatensystem abgibt. Die Erstellung des Plots wird mit der Funktion `figure` vorgenommen (Z.86f.). Hier können unter anderem Größe und Hintergrundfarbe des Plots, sowie der zuvor erstellte Tooltip angegeben werden. Im Anschluss werden Dimensionen der X- und Y-Achse festgelegt. Da die Schallempfindung des menschlichen Ohrs im Durchschnitt zwischen 20Hz und 20kHz stattfindet, liegt die Dimension der X-Achse im Bereich von -25000(Hz) bis 25000(Hz). An dieser Stelle sei anzumerken, dass die Ausgabe einer Fouriertransformation auch negative Werte zulässt, da auch

negative Frequenzen zum Ausgangssignal und der mathematischen Berechnungen der Fouriertransformationen „passen“. Die Datenpunkte werden als kreisförmiger Punkt in einem zweidimensionalen Koordinatensystem angegeben. Der Parameter `source` gibt die zuvor thematisierte Datenquelle an.

Einschub:

Die fouriertransformierten Daten geben den Ausschlag einer Frequenz in einem bestimmten Hertz-Bereich an. Diese liegen in sogenannten Buckets vor. Diese Buckets sind von der Sample-Rate und Größe der Datenmengen abhängig. Das allgemeine Beispiel gibt eine Sample Rate von 44,1kHz und eine Größe von 5000 Samples an. Daraus können Frequenz-Buckets berechnet werden, die in Korrelation zu den berechneten Daten anhand der Position im Ausgabearray stehen. Dabei enthält die Ausgabe sowohl positive als auch negative Zuordnungen.

```
freq_buckets = list()
for i in range(0, int(threshHold/2)):
    freq_buckets.append(i * sample_rate / threshHold)

for i in range(int(-threshHold/2), 0):
    freq_buckets.append(i * sample_rate / threshHold)
```

Die erste Hälfte der Daten enthält die positiven Frequenzeinträge. Analog werden negative Frequenz-Buckets in der zweiten Hälfte angegeben. Die Berechnung eines Buckets ist definiert durch: $i * sample_rate / size$. Die Variable `i` beschreibt den fortlaufenden Index eines solchen Buckets und `size` die Größe der Eingabedaten, was in diesem Kontext mit `threshHold` gleichzustellen ist. So würde die Berechnung für die Beispielindizes 100 und 101 die Frequenzen 882 bzw. 890,2 ergeben. Hier wird ein sehr wichtiger Punkt klar: Die Größe einer Fouriertransformation bestimmt zugleich deren Genauigkeit. Mit dem oben angegebenen Beispiel kann der Frequenzbereich zwischen 882 und 890,2 bei einer Größe von 5000 Eingabesamples niemals abgebildet werden.

Die Frequenz-Buckets werden im Plot Koordinatensystem in X-Richtung angegeben. Auf der Y-Achse stehen die errechneten Ausschläge für die entsprechenden Frequenzen. Die Übertragung in das Plot-Koordinatensystem erfolgt mit dem zuvor erwähnten JavaScript Adapter aus Abbildung 22.

```

58 adapter = CustomJS(code="""
59     const result = {x: [], y: []}
60     const freq_buckets = cb_data.response.frequency_buckets
61     const magnitudes = cb_data.response.magnitudes
62     const size = cb_data.response.size
63     const channels = cb_data.response.channels
64
65     for (let i = 0; i < channels; i++)
66     {
67         for (let j = 0; j < size; j++)
68         {
69             result.x.push(freq_buckets[j])
70             result.y.push(magnitudes[i][j])
71         }
72     }
73
74     return result
75 """)

```

Abbildung 23: JavaScript-Code zur Erstellung der Datenarrays

Der JavaScript-Code des Adapters wird in Form eines Strings erstellt, der dem Browser zum Zeitpunkt der Verbindung übermittelt wird. Dieser wird ausgeführt, sobald Daten über die URL `http://localhost:port/data` empfangen wurden. Die empfangenen Daten werden in Koordinatenpunkte des Plots umgewandelt und der Visualisierung angefügt. Der Python Webserver übermittelt in Form von JSON-Strings. Auf die Browser-Anfrage auf o.g. URL reagiert die folgende Implementierung:

```

@app.route('/data', methods=['GET', 'OPTIONS', 'POST'])
@crossdomain
def data():
    return json.dumps({'frequency_buckets': freq_buckets, 'magnitudes': list(
        magnitudeHolder),
        'channels': channels, 'size': threshHold})

```

Die Annotation `@app.route` gewährleistet, dass die definierte Funktion auf Anfragen der URL `/data` ausgeführt wird. Eine weitere Annotation `@crossdomain` liefert die hierfür nötigen Response-Header. Der Server schickt dem Browser Informationen über die Frequenz-Buckets, die dazu berechneten fouriertransformierten Daten (aufgeteilt auf die Kanäle) sowie die wichtigen Angaben für Anzahl der Kanäle und Größe der Samples. Abhängig vom angegebenen Polling-Intervall werden so die Daten ständig an den Browser übertragen. Das Python-Modul kann von beiden AudioTracer-Modulen Daten empfangen und verarbeiten. Das gleichzeitige Empfangen von beiden Endpunkten ist dabei nicht möglich.

6 Ergebnisse

Dieses Kapitel listet einige Testreihen mit verschiedenen Audiodateien auf. Für jede Audiodatei werden Tests mit zwei verschiedenen Schwellwerten an Eingabesamples und diese jeweils für beide AudioTracer-Module wiederholt. Die Testreihen sind wie folgt einzuordnen:

Testreihe	#Samples	#Kanäle	Modul(AudioTracer)
1	5000	2	C
2	5000	2	Python
3	25000	2	C
4	25000	2	Python
5	5000	4	C
6	5000	4	Python
7	25000	4	C
8	25000	4	Python

Die Testreihen für die Module werden in den nächsten beiden Sektionen getrennt dargestellt. Es werden zwei verschiedene Audiodateien mit zwei bzw. vier Audiokanälen verwendet. Die Dateien bestehen aus je einer Sinuskurve für jeden Audiokanal mit bekannter Frequenz. Die Frequenzen sind in folgender Übersicht aufgelistet:

Datei/Kanal#	1	2	3	4
2channelKnownFreqTest.wav	300Hz	40Hz	-	-
4channelKnownFreqTest.wav	30Hz	100Hz	160Hz	210Hz

6.1 AudioTracer C

Die Auswertung für das AudioTracer(C)-Modul thematisiert die Testreihen 1, 3, 5 und 7. Dabei werden die Reihen mit gleicher Anzahl von Audiokanälen nebeneinander abgebildet. Hierbei sei zu bemerken, dass die Ergebnisse eine Momentaufnahme einer Live-Visualisierung darstellen. Jeweils ein Block der fouriertransformierten Daten der Größe #Samples wird hier abgebildet.

Testreihe 1 und 3 -

Testreihe/erwartet	300Hz	40Hz
1	299,88Hz, -299,88Hz	44,1Hz, -44,1Hz
3	299,88Hz, -299,88Hz	38,808Hz, -38,808Hz

Die negativen Frequenzen werden zukünftig nicht mehr mit aufgezählt, da diese rechnerisch durch die Fouriertransformation gleich sind. Aus Gründen der Vervollständigung wurden diese hier mit angegeben und sind implizit bei den restlichen Testreihen zu erwarten.

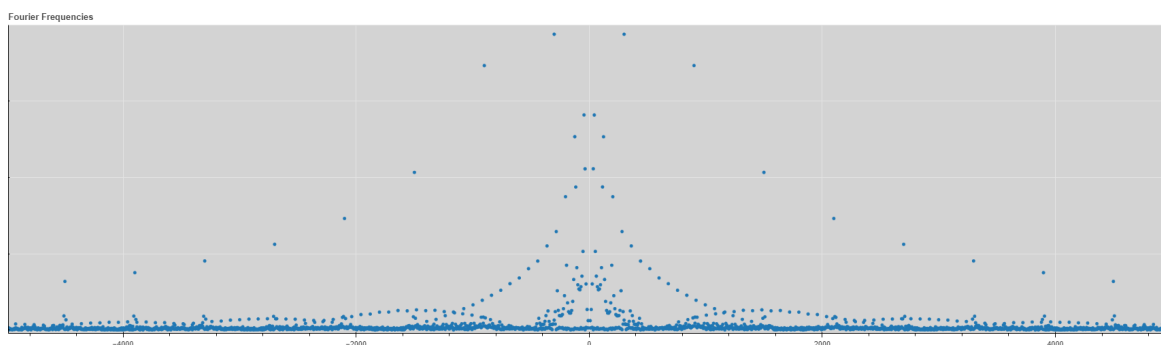


Abbildung 24: Test 1

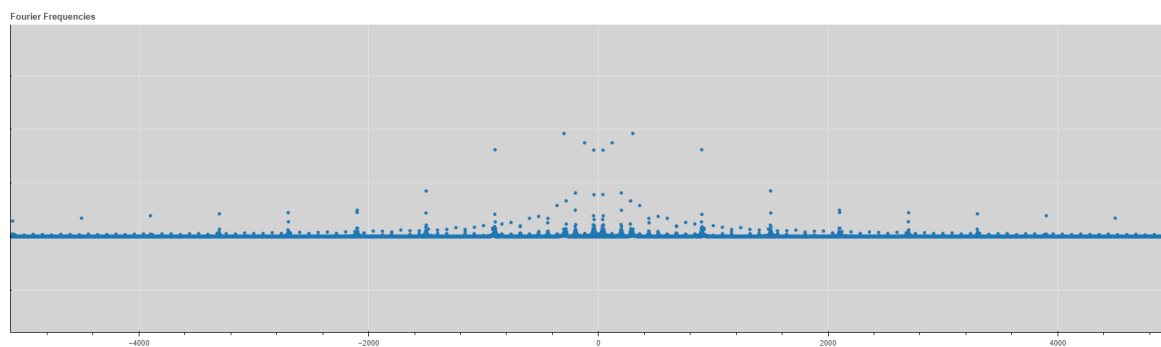


Abbildung 25: Test 3

Testreihe 5 und 7 -

Testreihe/erwartet	30Hz	100Hz	160Hz	210Hz
5	26,46Hz	97,02Hz	158,76Hz	211,68Hz
7	29,988Hz	100,548Hz	160,524Hz	209,916Hz

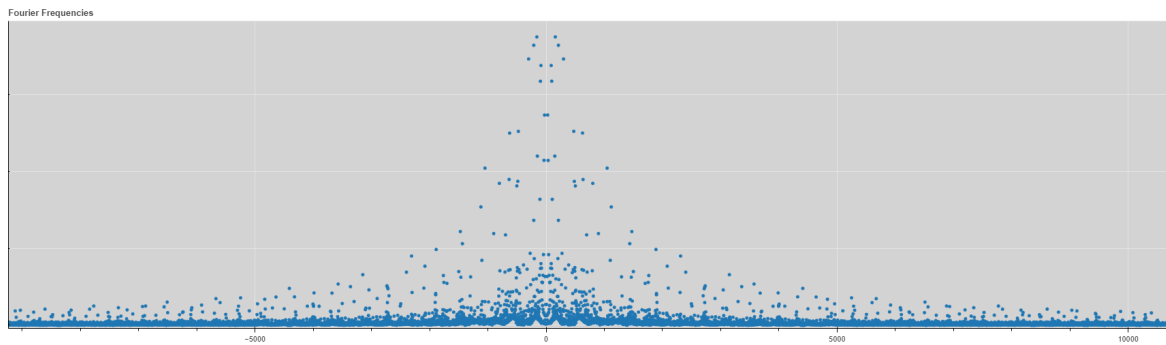


Abbildung 26: Test 5

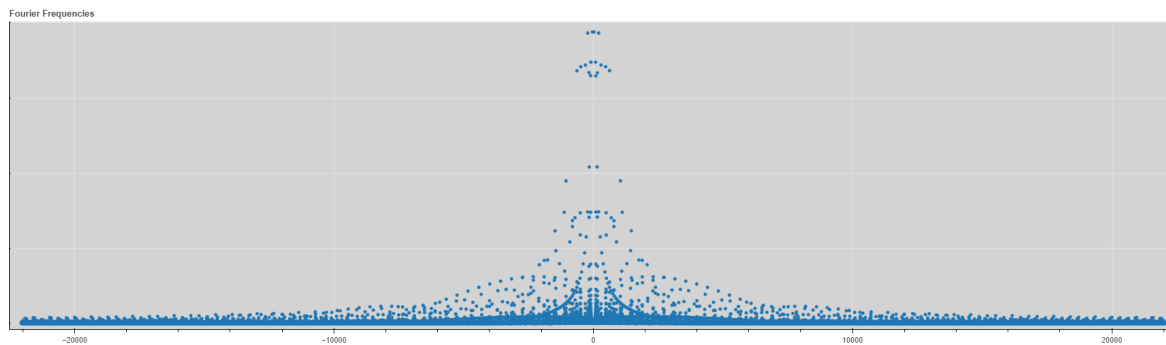


Abbildung 27: Test 7

6.2 AudioTracer Python

Testreihe 2 und 4 -

Testreihe/erwartet	300Hz	40Hz
2	299,88Hz	44,1Hz
4	299,88Hz	40,572Hz

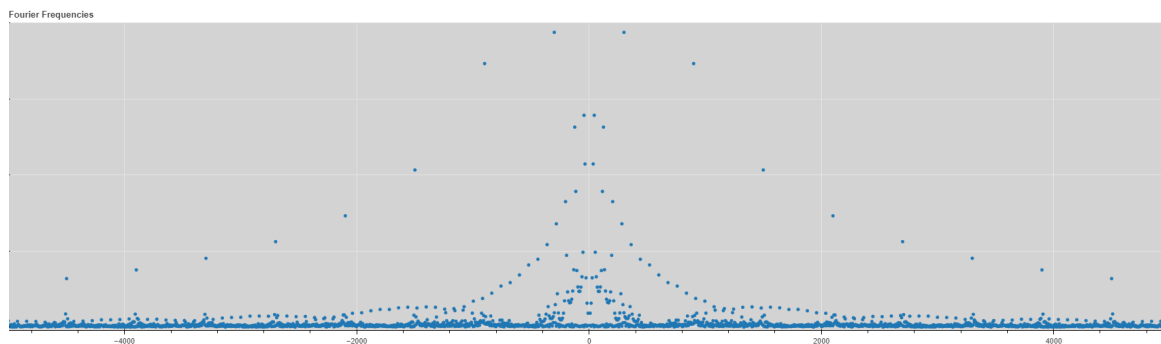


Abbildung 28: Test 2

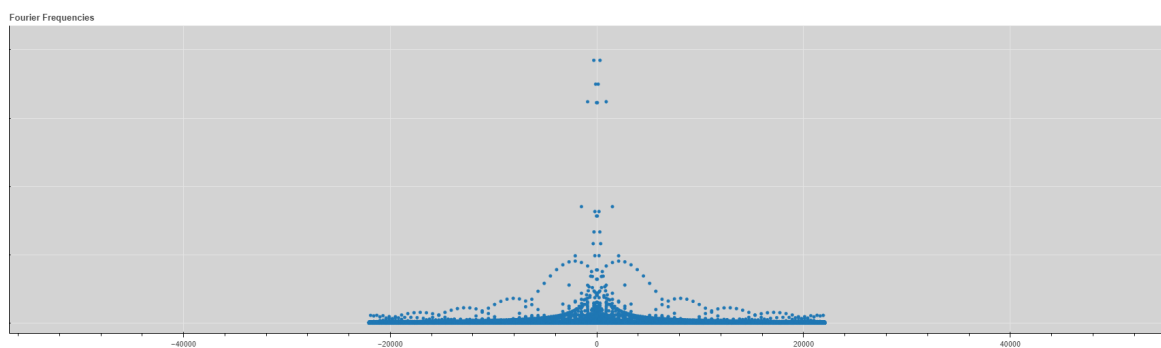


Abbildung 29: Test 4

Testreihe 6 und 8 -

Testreihe/erwartet	30Hz	100Hz	160Hz	210Hz
5	26,46Hz	97,02Hz	158,76Hz	211,68Hz
7	29,988Hz	100,548Hz	160,524Hz	209,916Hz

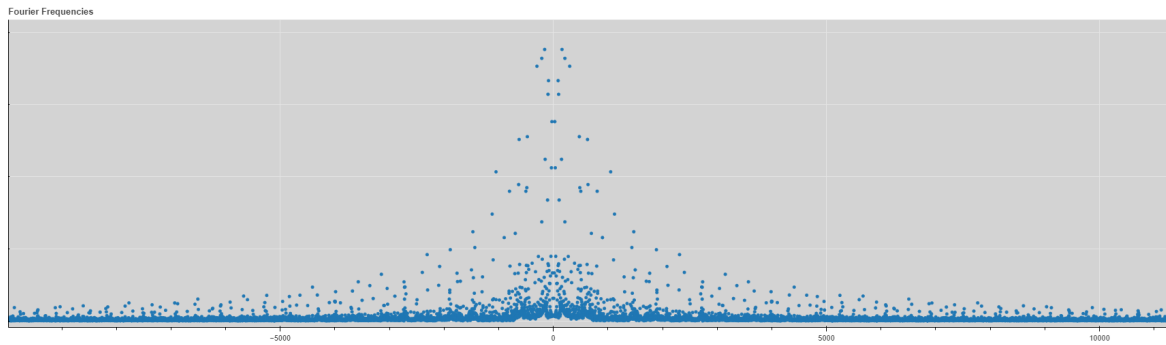


Abbildung 30: Test 6

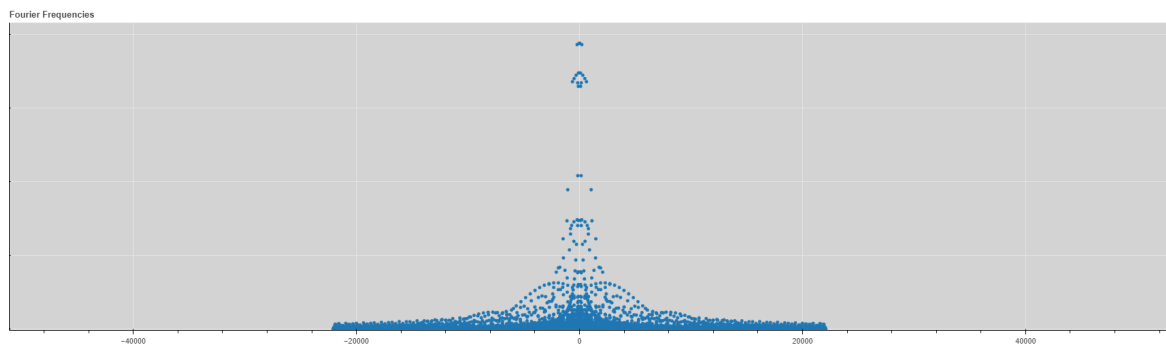


Abbildung 31: Test 8

6.3 Vergleich der Testreihen

Das Ziel der Anwendung einer Fouriertransformation ist eine möglichst genaue Extraktion der einzelnen Frequenzen, die zum angegebenen Teilbereich gehören. Erkennbar ist, dass die ermittelten Frequenzen nicht genau die erwarteten Hertz-Zahlen erreichen. Ein genereller Trend ist bei den Testreihen 5, 7 und 6, 8 mit vier Audiokanälen zu erkennen. Hier werden die Ergebnisse mit steigender Sample Anzahl genauer. Die Abweichungen liegen den Reihen 7 und 8 bei 0,012 bis 0,84 Hz. Die Testreihen 5 und 6 weisen einen Fehlerbereich von 1,24 bis 3,54Hz auf, was die Eingangsthese unterstützt.

Bei den Testreihen mit nur zwei Audiokanälen ist eine ähnliche Tendenz zu erkennen, jedoch treten hier folgende Anomalien auf: Der mit 300Hz zu erwartende Wert weist bei allen vier teilhabenden Testreihen (1, 3, 2 und 4) denselben Wert von 299,88Hz auf. Bei zu erwartenden 40Hz im zweiten Audiokanal liegen die Testreihen mit 5000 Audiosamples mit 4,1Hz deutlich über denen mit 25000 Audiosamples (1,192Hz und 0.572Hz). Außerdem sind die Ergebnisse zwischen C- und Python-Modul unterschiedlich, was bei den Testreihen mit vier Audiokanälen nicht der Fall ist. Die Annahme liegt nahe, dass bedingt durch die Momentaufnahme der Ausgabe ein „ungünstiger“ Moment aufgenommen wurde und somit die Daten ungenau werden. Der Unterschied zwischen zwei Frequenzen fällt bereits bei 1Hz Differenz durch genaues Hinhören auf.

Allgemein sticht jedoch heraus, dass die Erhöhung der Eingabesamples für eine genauere Berechnung der Fourierfrequenzen sorgt.

7 Teilprojekt: „Audioparcours mit CUDA“

Das Teilprojekt Audioparcours mit CUDA stellt ein Anwendungsbeispiel für CUDA in der Signalverarbeitung dar. Der Grundgedanke zur Implementierung dieses Anwendungsfalls ist, ein Eingangssignal in Form einer WAV-Datei mithilfe der Shader-Kerne über die CUDA-Schnittstelle zu manipulieren, sodass ein Effekt der Positionsveränderung zur gegebenen Soundquelle entsteht. Durch Nutzereingaben können die Positionsdaten in Rotation und Entfernung zum Startpunkt verändert werden. Die Herausforderung ist dabei unter anderem eine Audioquelle zu manipulieren, die gleichzeitig auf einem Wiedergabegerät abgespielt wird. Die Einzelheiten zu diesen Techniken werden im folgendem Kapitel offengelegt.

7.1 Überblick

In der Ausgangssituation steht die schon im Projekt AudioTracer verwendete WAV-Datei `2channelKnownFreqTest.wav` zur Verfügung. Analog zum Dateinamen sind zwei Audiokanäle (links und rechts) vorhanden, auf denen jeweils ein harmonisches Audiosignal wiedergegeben wird. Auf dem linken Kanal ist eine Sinuswelle mit einer Frequenz von 300Hz gespeichert. Der rechte Kanal hält eine Sinusfrequenz von 30Hz. Diese Datei bietet sich für Testzwecke an, da auf jedem Kanal nur eine gleichbleibende Frequenz wiedergegeben wird und somit leicht zu unterscheiden ist.

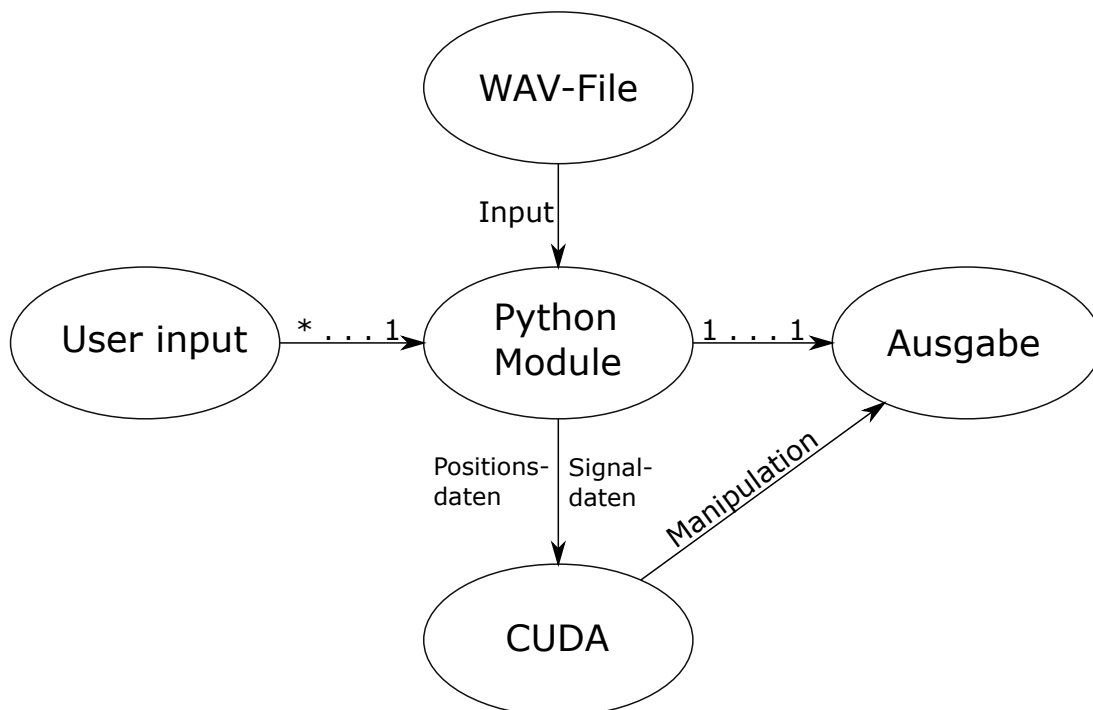


Abbildung 32: Überblick Teilprojekt Audioparcours mit CUDA

In Abbildung 32 ist eine grobe Übersicht auf das implementierte Audioparcours-Projekt zu sehen. Der erste Schritt ist hier, die WAV-Daten in einer Datenstruktur im erstellten Python Modul verfügbar zu machen. Im Gegensatz zum Projekt AudioTracer liegen die Audiodaten in diesem Fall statisch vor. Es wäre durch leichte Erweiterungen möglich, die beiden Projekte AudioTracer und Audioparcours an dieser Stelle miteinander zu koppeln (dazu mehr im Kapitel 10). Das Python Modul ist für die Initialisierung des CUDA-Kontexts und das Verarbeiten der Nutzereingaben zuständig. Außerdem wird die Ausgabe auf einem Audiogerät hier realisiert. Als Nutzereingaben werden Tasten des Pfeiltastenblocks ('up', 'down', 'left', 'right') akzeptiert. Die horizontalen Eingaben verändern eine Rotation und die vertikalen die Entfernung vom Mittelpunkt.

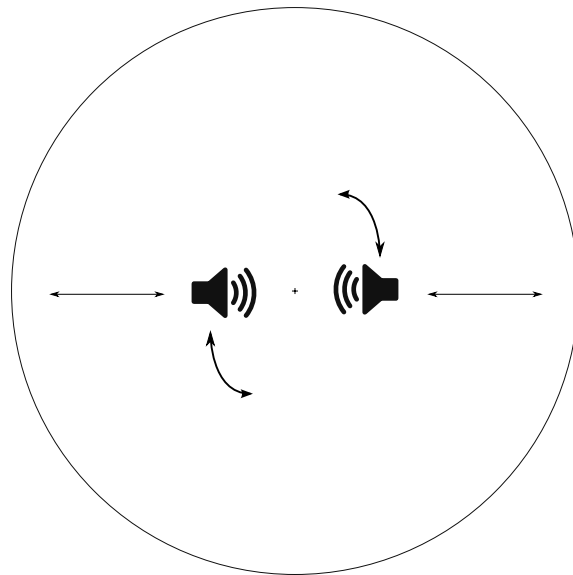


Abbildung 33: Positionsveränderung Audioparcours

Abbildung 33 zeigt das grundsätzliche Prinzip der Positionsveränderung in einer Art uhrförmigen Struktur. Die Positionsangaben sind relativ zum Mittelpunkt dieser „Uhr“ zu betrachten. In dieser Analogie würden die Zeiger in ihrer Rotation und Länge respektive die Position der beiden Lautsprecher bestimmen. Diese Werte werden vom ausführenden Benutzer gesteuert und können während der Wiedergabe der Audiodatei verändert werden. Die Manipulation des Eingabesignals wird passend zum Kontext dieser Arbeit in einem CUDA-Kontext realisiert. Bisher wurde die Implementierung der CUDA-Schnittstelle für die Programmiersprache C näher erläutert. Da das hier vorgestellte Modul in Python implementiert wurde, sind ein paar Einzelheiten vorher klarzustellen.

Python hat keine direkte Schnittstelle zur Grafikkarte, kann jedoch Daten vermitteln und wieder extrahieren. Der individuell auszuführende Code auf den Shader-Kernen wird in einem sogenannten **SourceModule** in einem String angegeben. Der String muss eine lauffähige Methode in C mit korrekter Syntax repräsentieren. Das ist die aus der

Sektion 5.3.2 bekannte global deklarierte Methode, welche im Host und Device Kontext referenzierbar ist. Das Python Objekt `SourceModule` gibt eine Referenz zu dieser Methode zurück, welche zu einem späteren Zeitpunkt direkt vom Python-Modul ausgeführt werden kann. Die Wiedergabe eines Audiosignals wird mit dem Paket **Sounddevice** realisiert. Durch dieses Paket besteht die Möglichkeit, ein Stream-Objekt zu erstellen, das die rohen Daten direkt via einer Callback-Methode auf einem bestimmten Audiogerät abspielt. Die Callback-Methode ist die Schlüsselstelle, an der die Manipulation der WAV-Daten eingesetzt werden kann, da ein solcher Audiostream nach der Wiedergabe einer bestimmten Anzahl an Samples - hier Blockgröße genannt - die Callback-Methode nach Definition aufruft, damit dort der nächste Block bestimmt werden kann. Die zwei Techniken der Sound-Manipulation und der Wiedergabe sind dabei nicht abhängig voneinander. Dies ist immens wichtig, da ansonsten in der Zeit, in der der CUDA Teil die Sounddaten manipuliert werden, Verzögerungen in der Wiedergabe zu hören sein könnten. Im schlimmsten Fall würde das Programm hier mit einem Fehler stoppen. Dieses Problem wird dadurch gelöst, dass in der Übergangszeit das zuvor manipulierte Signal wiedergegeben wird, bis die nächste Iteration der CUDA-Routine abgeschlossen ist. Der Ablauf der internen Strukturen des Audioparcours-Projekts sind im nachfolgenden Sequenzdiagramm 34 visualisiert.

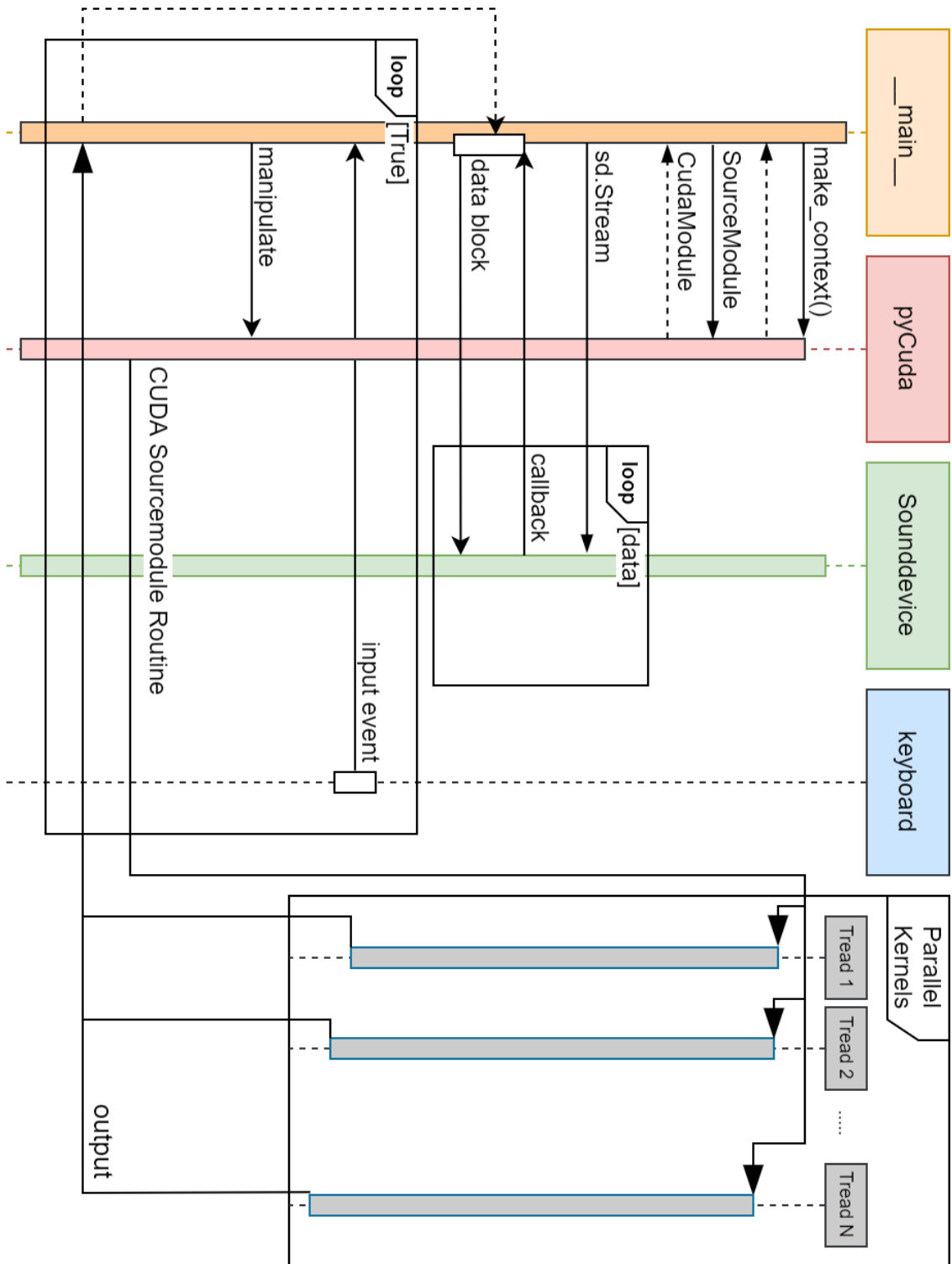


Abbildung 34: Sequenzdiagramm Audioparcours

7.2 Code Architektur

Im folgenden Kapitel werden Einzelheiten zur Implementierung offengelegt. Vor allem die in Sektion 7.1 beschriebenen Techniken werden hier genauer betrachtet. Zunächst werden die wichtigsten importierten Pakete zur Realisierung des Audioparcours Projekts aufgelistet und deren Funktion im Programm angegeben.

```
1 import math
2 import numpy as np
3 import scipy.io.wavfile as wavfile
4 from pycuda.compiler import SourceModule
5 import pycuda.driver as drv
6 import sounddevice as sd
7 import keyboard
8 from dataclasses import dataclass
```

Abbildung 35: Verwendete Pakete im Projekt Audioparcours

In Zeile 3 wird das Paket `scipy.io.wavfile` eingebunden. Dies erlaubt eine leichte Einbindung von WAV-Dateien in das Python-Projekt. Im AudioTracer-Projekt hat diese Funktionalität die Klasse `WAVLoader` übernommen. Zeile 4 bindet das wichtigste Paket `SourceModule` aus dem PyCUDA Projekt ein. Ein `SourceModule` erlaubt es, validen C-Code als Funktion in Python zu verwenden. Wie in Sektion 7.1 beschrieben wurde, wird diese Schnittstelle benötigt, da es sich bei CUDA um eine C-Library handelt und somit Python nicht nativ unterstützt. Das Paket `pycuda.driver` aus Zeile 5 ist für die generelle Initialisierung des CUDA-Kontexts zuständig und steuert zum Beispiel die Verteilung eines Prozesses auf die CUDA-Kernels und die Auswahl der GPU, auf der diese Art von Rechnungen durchgeführt werden sollen. In Zeile 6 wird das Paket `sounddevice` eingebunden. Darüber wird der Zugriff auf ein Audio-Ausgabegerät möglich. Die Nutzereingaben erfolgen über die Tastatur. Die Schnittstelle, um solche Eingaben abzufangen, ist das in Zeile 7 eingebundene Paket `keyboard`. Zeile 8 bindet das Paket `dataclass` ein. Eine `dataclass` ist eine in Python interpretierte Klasse oder Struct um z.B. für einen bestimmten Kontext individuelle Attributklassen zu erstellen.

```
11 @dataclass
12 class Position:
13     rotation: int
14     distance: float
```

Abbildung 36: Python: dataclass

In diesem Fall wird eine `dataclass` verwendet, um die Positionsdaten zu speichern und für den weiteren Verlauf verfügbar zu machen. Dabei muss die Annotation `@dataclass` verwendet werden, um Built-in Getter- und Setter-Methoden verfügbar zu machen. Zwei

Variablen werden analog zur Beschreibung in 7.1 angelegt. Der Integer-Wert **rotation** gibt die Rotation in eine bestimmte Richtung an. In diesem Projekt wird einem Rotationswert größer dem Wert 0 eine Rotation nach rechts zugeordnet. Analog bildet eine negative Rotation eine Linksdrehung ab. Die float Variable **distance** gibt die simulierte Distanz zwischen Soundquelle und Hörer an. Dies sind die wichtigsten Pakete die benötigt werden, um die beschriebene Funktionalität zu implementieren.

Die nächsten drei Codeblöcke implementieren das Einlesen des WAV-Files, das Erstellen des CUDA-Kontexts und die Initialisierung von wichtigen später verwendeten Variablen.

```
18 fs_rate, signal = wavfile.read("../2channelKnownFreqTest.wav")
19 num_channels = 2
20 original = signal.copy()
21 transitionHelper = signal.copy()
```

Abbildung 37: Audioparcours: Einlesen der WAV-Datei

Im ersten Codestück 37 wird in Zeile 18 mithilfe des zuvor eingebundenen Pakets **scipy.io.wavfile** die Audiodatei eingelesen und als Array-Struktur verfügbar. Die Methode **read** erwartet eine Angabe des Pfades zur WAV-Datei und gibt neben den Kanaldaten die Sample Rate zurück, die im späteren Verlauf verwendet wird. Die Angabe der Anzahl der Audiokanäle muss manuell in der darauffolgenden Zeile durchgeführt werden. Zeile 20 legt eine Kopie des Originalsignals an, auf der die späteren Manipulationen durchgeführt werden. Die Variable **transitionHelper** ist eine weitere Kopie, die als Überbrückung des asynchronen Vorgangs der Audiomaniplulierung verwendet wird. Mit der Einbindung von CUDA-Operationen kann nicht davon ausgegangen werden, dass die aktuell wiedergegebene Audiodatei weiterhin abgespielt werden kann, während sie in Verwendung im CUDA Teil des Programms ist.

```
25 drv.init()
26 print("GPUs available ", drv.Device.count())
27 device = drv.Device(0)
28 ctx = device.make_context()
```

Abbildung 38: Audioparcours: Erstellung des CUDA-Kontext

Der zweite Codeblock 38 initialisiert die Verwendung der CUDA-Schnittstelle. Dies wird durch die Erstellung eines CUDA-Kontexts realisiert. Ein CUDA-Kontext ist ein bestimmter Zustand, in der sich die CUDA-Runtime API für einen Programmablauf befindet. Dieser Kontext muss konsistent bleiben, beispielsweise ist dieser nur auf dem Thread verfügbar, auf dem er erstellt wurde. Zeile 25 initialisiert das Device-API-Interface gefolgt von der Auswahl der CUDA-fähigen GPU in Zeile 27. Der CUDA-Kontext wird mit der **make_context()** der Device API erstellt (Z. 28).

```

31 index = 0 # idx to track upcoming audio samples
32 block_size = 1024 # random picked power of 10
33 blockStream = False # bool to block using currently manipulated stream
34 MAX_CUDA_KERNELS = 1024
35 p = Position(0, 0)

```

Abbildung 39: Audioparcours: Globale Variablen

Im dritten Block der Vorbereitungen zur Implementierung der Wiedergabe- und Manipulationslogik werden wichtige globale Variablen festgelegt. Die Variable `index` in Zeile 31 wird dafür verwendet, den korrekten Ablauf bei der Wiedergabe der Audiodatei einzuhalten, der via einer Callback-Funktion - die im Laufe dieses Kapitels noch aufzuführen ist - bestimmt werden kann. In Zeile 32 wird eine `block_size` angegeben. Diese spiegelt die Länge der Audiosamples wieder, die zwischen zwei aufeinander folgenden Callback-Funktionsaufrufen wiedergegeben werden und wird zufällig gewählt auf den Wert 1024 gesetzt. Die Variable `blockStream` verhindert, dass die Callback-Funktion den gerade manipulierten Datensatz als Soundquelle auswählt. Während des Manipulationsvorgangs wird das zuletzt veränderte Soundsignal wiedergegeben, bis die CUDA-Routine erfolgreich beendet wurde. Dies verhindert eine Latenz beim abspielen. Für den Benutzer ist eine so künstlich hergestellte Verzögerung bis der neue Datensatz bereit ist, nicht erkennbar. Aus 5.3.2 und 4.3 ist bekannt, dass in der Testumgebung eine Geforce GTX 1080 GPU verwendet wurde. Die maximale Anzahl der aktiven Shader-Kerne in einem CUDA-Grid liegt bei 1024 und wird somit in Zeile 34 analog definiert. Zeile 35 initialisiert die zuvor erstellte `dataclass Position` mit dem Wert 0 für beide Attribute.

Mit diesen Vorbereitungen können nun die weiteren Implementierungen vorgenommen werden. Es wurde erwähnt, dass seitens CUDA ein so genanntes `SourceModule` verwendet wurde, um eine Methode in validem C-Code zu erstellen, diese dann in Python zum Aufruf verfügbar wird.

Im abgebildeten Code von Abbildung 40 wird ein solches `SourceModule` im konkreten Fall sichtbar. Hier wird die global deklarierte Methode `manipulate` definiert. Die Signatur dieser Methode ist in Zeile 66 zu sehen. Im Parameter `int *dest`, hier als Integer Pointer deklariert, werden die manipulierten Sampledaten gespeichert. Der Integer Pointer `int *src` dient als Datengrundlage und bildet in den einzelnen Durchläufen die Originaldaten in eindimensionaler Form ab. Diese Technik der Abflachung von Arrays wurde bereits im Projekt AudioTracer verwendet, da das Arbeiten mit mehrdimensionalen Arrays generell aufgrund der höheren Komplexität in der Speicherallokation nicht empfohlen wird [15]. Der Parameter `int *rotation` gibt einen Rotationswert im Wertebereich $[0, 180[$ an. Dieser Wert ist maßgeblich an der Verteilung der Audiosignale auf die anderen Soundkanäle zuständig, denn je nach Rotation werden Teile von benachbarten Kanälen aufaddiert, um einen adäquaten Effekt der Positionsabhängigkeit zu erhalten. Analog bestimmt der Parameter `float *distance` die simulierte Entfernung zur Soun-

```

65 mod = SourceModule("""
66     __global__ void manipulate(int *dest, int *src, int &rotation, float &distance,
        int &parts, int &numChannels)
67     {
68         const int idx = threadIdx.x;
69         float modRot = rotation / 180.f;
70         for (int i = idx * parts; i < idx * parts + parts; i+=numChannels)
71         {
72             for (int ch = 0; ch < numChannels; ch++)
73             {
74                 int chIdx = i + ch;
75
76                 dest[chIdx] = (src[chIdx] - src[chIdx] * modRot +
77                     src[i + abs((chIdx-1) % numChannels)] * modRot) / numChannels;
78
79                 if (distance > 1 || distance < -1)
80                     dest[chIdx] = dest[chIdx] / distance;
81             }
82         }
83     }
84 """)

```

Abbildung 40: Audioparcours: SourceModule

dquelle. Je höher der Wert, desto - einfach ausgedrückt - leiser müssen die einzelnen Soundkanäle werden. Da eine Audiodatei meistens eine höhere Anzahl von Audiosamples aufweist, als CUDA-Kernel verfügbar sind, werden alle Audiosamples auf die Kernels aufgeteilt, d.h. jeder CUDA-Kern berechnet einen zusammenhängenden Teilbereich für die Ausgabe. Diese Aufteilung wird durch den Parameter `int *parts` gesteuert. Die Variable `int *numChannels` wird für die Iteration der einzelnen Audiokanäle benötigt, sowie der Berechnung des Mittels mehrerer addierter Samples, beispielsweise wenn ein Teile eines Channels abhängig von der Rotation in einem anderen zu hören sind. Auffällig ist hier, dass alle Parameter in Form von Referenzen übergeben werden. Dies war notwendig, da hier ein pass-by-value anders als in offiziellen Dokumentationen [16] nicht möglich ist.

In Zeile 68 wird der Index des Kernel-Threads, der eindeutig für jeden ausgeführten Thread verfügbar ist, bestimmt. Dieser wird verwendet, um den „Zuständigkeitsbereich“ eines Threads für die Manipulation zu berechnen. In der darauffolgenden Zeile wird eine Variable `float modRot` in Abhängigkeit der Rotation geteilt durch den Maximalwert der Rotation bestimmt. Dieser muss bei der Multiplikation mit einem Audiosample zwischen 0 und 1 liegen, da anderenfalls bei größeren Rotationswerten die Wiedergabe an diesen Stellen übersteuern würde. Zeile 70 beschreibt den Laufbereich einer Schleifeniteration, wobei die zuvor beschriebenen Parameter `idx` und `parts` benötigt werden,

um geeignete Grenzbereiche zu kalkulieren. Die Iteration wird jeweils um die Anzahl der Audiokanäle erhöht. Innerhalb dieser Anweisung ist eine weitere Iteration implementiert (Zeile 72-81). Die Hauptaufgabe in diesem Kontext ist die Bestimmung der Ausgabesamples im Array `dest`. Der aktuelle Sample Index wird dem Integer Feld `int chIdx` zugewiesen. Das neue Ausgabesample berechnet sich durch die Annahme, dass es sich in diesem Anwendungsfall beispielsweise nach Abbildung 33 um eine menschliche Person positioniert im Mittelpunkt zwischen zwei Audiokanälen handelt. Bei einer Rotation in eine Richtung würde ein Sample, das nur aus dem rechten Audiokanal zu hören ist, zunächst bei fortlaufender Rotation in dieselbe Richtung leiser zu hören sein und schließlich nur auf dem linken Kanal. Dasselbe muss für die Rückrichtung vom linken auf den rechten Kanal gelten. Nach dieser Analogie würde in Zeile 76f. das aktuell betrachtete Sample mit der Teilrechnung `src[chIdx] - src[chIdx] * modRot` leiser werden. Dieses wird dann mit einem Anteil des vorherigen Audiokanals addiert `+src[i + abs(chIdx-1) % numChannels]`. Dabei wird durch die Verwendung der Methoden `abs()` und dem Modulo-Operator sichergestellt, dass keine Werte außerhalb des zulässigen Wertebereichs adressiert werden. Wäre der aktuell betrachtete Audiokanal 1 angenommen, würde ein Teil des `abs(0-1) % 2 = 1`, also des zweiten Kanals, dazu addiert. Die Addition von Audiokanälen kann so jedoch nicht stehen gelassen werden. Das Ergebnis muss zwingend durch die Anzahl der Kanäle geteilt werden, da die anteiligen Frequenzen sonst nicht mehr stimmen würden. Nach diesem Prinzip werden reihum alle Audiokanäle manipuliert. Die Berechnung der Distanz ist folgend weniger kompliziert (Z. 79f.). Liegt der Wert der Variable `distance` nicht zwischen -1 und 1 wird der aktuelle Kanal durch diesen Wert geteilt. Dadurch verändert sich die Amplitude der anteiligen Sinuswellen und es entsteht ein Effekt der Entfernung zum Wiedergabegerät in Form von einer geringeren Lautstärke an der Stelle.

Das hier beschriebene `SourceModule` wird seitens Python durch die Zuweisung `manipulate = mod.get_function(„manipulate“)` zugreifbar und kann mit entsprechenden Parametern auf der Grafikkarte ausgeführt werden.

Als nächster Schritt muss ein Audiostream erstellt werden, der zu bestimmten Zeitpunkten verändert werden kann. Das eingebundene Paket `Sounddevice` bietet eine solche Funktionalität mithilfe der Methode `Stream(...)` an, deren Einzelheiten im folgenden diskutiert werden.

```
87 stream = sd.Stream(device=sd.default.device, samplerate=fs_rate, channels=
    num_channels,
88                      callback=callback, blocksize=block_size, dtype='int16')
```

Abbildung 41: Audioparcours: Sounddevice Stream

Die Methode `Stream` vom Paket `sounddevice` erwartet im Wesentlichen vier Parameterangaben. Der Parameter `device` gibt das Ausgabegerät an und wird in diesem Fall mit dem Standardwiedergabegerät `sd.default.device` beschrieben. Die Sample-Rate (`samplerate`) wurde beim Einlesen des WAV-Files zurückgegeben und in der Variable `fs_rate` gespeichert. Die Sample Rate bestimmt, wie viele Samples in einer Sekunde abgespielt werden. Im Parameter `channels` wird die Anzahl der Audiokanäle an den Audiostream übergeben. Außerdem wird eine Angabe erwartet, um welchen Integer-Typ es sich bei den Sampledaten handelt `dtype`. Der letzte Parameter `callback` ist im Kontext dieses Projekts ein sehr wichtiger Meilenstein.

```
38 def callback(in_data, out_data, frames, time, status):
39     global index, blockStream
40
41     if index > len(signal) - 1:
42         index = 0
43
44     block_increase = block_size
45
46     if index + block_increase > len(signal) - 1:
47         block_increase = len(signal) - index
48
49     next_block = np.zeros_like(out_data)
50     if blockStream:
51         next_block[:block_increase] = transitionHelper[index:index + block_increase]
52     else:
53         next_block[:block_increase] = signal[index:index + block_increase]
54
55     out_data[:] = next_block[:]
56     index += block_increase
```

Abbildung 42: Audioparcours: Sounddevice Callback-Methode

Es wird eine individuell erstellte Callback-Funktion erwartet, die nach einer bestimmten Menge an wiedergegebenen Samples aufgerufen wird. Diese Menge wird in dem Parameter `blocksize` gespeichert. Die Callback-Funktion erlaubt es in diesen spezifischen Momenten, die aktuelle Wiedergabe zu manipulieren. Die Manipulation muss zwangsweise über die Callback-Methode durchgeführt werden, da dadurch gewährleistet wird, dass die korrekte Position der Wiedergabe eingehalten wird. Außerdem würden bei einer direkten Überschreibung des Audiostreambuffers Latenz und Störgeräusche entstehen.

Die Callback Methode ist in Abbildung 42 aufgelistet. Die Methode erwartet die Angabe von fünf Parametern:

- **in_data**: Dieser Parameter spiegelt die aktuell vom Audiostream verwendeten Audiosamples als mehrdimensionales Array ab. Hier findet dieser keine Verwendung, da die Wiedergabe zu keinem Callback-Zeitpunkt von den Eingabesamples abhängig ist.
- **out_data**: Dies ist das Ausgabearray. Alle Veränderungen treten ab dem nächsten wiedergegebenem Block in Kraft. Dieser Parameter dient als Ansatzpunkt für die Manipulation der Signaldaten. Der nächste Block der Wiedergabe wird aus dem reorganisierten Ergebnis der CUDA-Ausgabe aus 40 bestimmt. Die CUDA-Ausgabe muss dafür vor dem Callback in einer passenden Array Struktur vorliegen.
- **frames**: Der Parameter `frames` gibt die Anzahl der Samples an, die zwischen dem aktuellen und vorherigen Callback wiedergegeben wurden. Dies findet in diesem Anwendungsfall keine Relevanz.
- **time** ist ein Objekt, das Timestamps über die Signalumwandlung von digital zu analog und analog zu digital für jeweils das erste Sample in in- und output Buffer.
- **status**: Gibt Fehler Flags zurück, wenn ein Buffer-Over- oder Underflow während der Wiedergabe festgestellt wird. Fehler werden in diesem Fall lediglich ausgegeben, treten in der Regel allerdings nicht auf.

In Zeile 39 der Callback-Methode (42) werden die zuvor angelegten globalen Variablen auch als solche deklariert. Python Variablen sind Scope gebunden, sodass ohne diese Deklaration die beiden Variablen `index` und `blockStream` im äußeren Scope unverändert bleiben würden. Erstere kann so in Zeile 62 korrekt inkrementiert werden. Zeile 46f. bewirkt, dass die Wiedergabe in einer Schleife abgespielt wird. Dies wurde als hilfreich befunden, um auch mit kürzeren Audiodateien zu arbeiten, deren Wiedergabe schnell beendet wird, ohne eine Möglichkeit für viele Nutzereingaben zu gewährleisten. In Zeilen 49 bis 53 wird die Länge des nächsten Outputbuffers evaluiert und erhält initial den Wert der globalen `block_size`. Es kann sein, dass die Länge des letzten Blocks im Audiostream kleiner ist als `block_size`. Für diesen Fall wird die entsprechende Differenz kalkuliert. Die Bestimmung der Samples des nächsten Blocks wird innerhalb der Zeilen

55 bis 59 durchgeführt. Prinzipiell wird das `Ausgabearray` in Form des `Ausgabearrays` mit Nullen gefüllt. Danach werden die neuen Samples kopiert. Die Bool'sche Variable `blockStream` wird verwendet um die eingangs erwähnte Problemstellung zu kompensieren, dass während der CUDA-Kontext aktiv Veränderungen am Audiosignal vornimmt, dieses nicht gleichzeitig als Input benutzt werden kann. In dieser Übergangszeit wird die aktuelle Wiedergabe zur Überbrückung fortgesetzt. Zuletzt werden die Sampledaten in das `Ausgabearray` kopiert.

An diesem Punkt sind der `Audiostream` erstellt, das `SourceModule` interpretiert und alle restlichen Vorarbeiten getroffen, um die Nutzereingaben und die damit verbundene CUDA-Routine in einer wiederkehrenden Schleifenoperation abzuarbeiten.

```

91     while True:
92         if keyboard.is_pressed('left'):
93             p.rotation = (p.rotation - 1) % 180
94         elif keyboard.is_pressed('right'):
95             p.rotation = (p.rotation + 1) % 180
96         elif keyboard.is_pressed('up'):
97             p.distance += 0.1
98         elif keyboard.is_pressed('down'):
99             p.distance -= 0.1
100        else:
101            continue

```

Abbildung 43: Audioparcours: Nutzereingaben

Nutzereingaben können über die Pfeiltasten getätigt werden. Die Methode `is_pressed` wird dabei verwendet, um zu überprüfen, ob ein jeweiliger Tastenanschlag getätigt wurde. Werden die Pfeiltasten links und rechts betätigt wird die Rotation um jeweils die Zahl 1 dekrementiert bzw. inkrementiert. Anschließend wird der Modulo Operator verwendet, damit die Rotation den Wert 180 nicht überschreitet. Im Prinzip kann hier eine Angabe in Grad angenommen werden. Bei vertikalen Pfeiltasteneingaben wird die Distanz de- bzw. inkrementiert. Sonstige Eingaben werden nicht unterstützt.

Erfolgt eine gültige Nutzereingabe, wird der hintere Teil 44 der while-Schleife ausgeführt, der vor allem die CUDA-Methode `manipulate` aufruft, die über das `SourceModule` Konstrukt erstellt wurde. Die Eingabeparameter müssen mit dem Präfix `drv.In` versehen werden. Dies löst eine korrekte Speicherreservierung seitens der GPU aus. Analog wird die Ausgabe im Python Kontext durch den Präfix `drv.Out` nach dem Abschluss der CUDA-Routine zugreifbar. Außerdem müssen Integer und Float Variablen in die entsprechenden C-Typen umgewandelt werden. Die Blockdimensionen werden nicht weiter auf mehrere Dimensionen aufgeteilt. Dies ist an dieser Stelle nicht nötig, da nur der Index des Threads für diesen Anwendungsfall benutzt wird (Zeile 111 - 114). In Zeilen 105 wird das zuletzt verwendete Signal in das Übergangsarray `transitionHelper` kopiert. In Verbindung mit der Variable `blockStream` wird dieses Signal ab hier wiedergegeben, bis

die CUDA-Operationen abgeschlossen wurden. Das Originalsignal wird in Zeile 107 zu einem eindimensionalen Array und in den passenden Datentyp dabei konvertiert. Nach der Manipulation müssen diese Dimensionen wiederhergestellt werden (Zeile 116). Das Ausgabearray wird zunächst mit dem Wert 0 gefüllt, um die Struktur der Eingabe beizubehalten. `parts` gibt die Anzahl der Teilstücke an, die Jeder CUDA-Kernel bearbeiten muss.

```
105     transitionHelper = signal.copy()
106     blockStream = True
107     signal_input = np.array(original.flatten(), dtype=np.int32)
108     dest = np.zeros_like(signal_input)
109     parts = math.ceil(len(signal_input) / MAX_CUDA_KERNELS)
110
111     manipulate(drv.Out(dest), drv.In(signal_input),
112                drv.In(np.int32(p.rotation)), drv.In(np.float32(p.distance)),
113                drv.In(np.int32(parts)), drv.In(np.int32(num_channels)),
114                block=(MAX_CUDA_KERNELS, 1, 1), grid=(1, 1))
115
116     signal = np.array(dest.reshape(len(original), num_channels), dtype='int16')
```

Abbildung 44: Audioparcours: Manipulation über CUDA

In diesem Python Quellcode des Audioparcours Projekts greifen verschiedene Faktoren ineinander mit teilweise asynchron laufenden Callback-Methoden und einem CUDA-Kontext der parallel Daten manipuliert, die im Callback verwendet werden. Diese Faktoren bringen eine nicht zu unterschlagene Komplexität mit sich, da viele logische Sprünge notwendig sind.

7.3 Ergebnisse

Die Ergebnisse des Audioparcours-Teilprojekts können nicht schriftlich festgehalten werden, da es sich hier um eine hörbare Manipulation einer WAV-Datei handelt. Das zugrundeliegende Programm liegt dieser Arbeit im Github-Repository [3] als Python-Skript `soundManipulator.py` vor und kann unter der Angabe einer Audiodatei im WAV-Format mit zwei Audiokanälen ausgeführt werden.

8 Skalierbarkeit

Die beiden Teilprojekte AudioTracer und Audioparcours wurden mit verschiedenen Testeingaben ausgeführt und entsprechende Ergebnisse bereits diskutiert. An diesem Punkt bleibt die Frage der Skalierbarkeit beider Teilprojekte unbeantwortet.

Das **AudioTracer** Projekt wurde analog zu Kapitel 6 in acht Testfällen getestet, jeweils vier für das C- bzw. Python. Die Anzahl der Samples und Audiokanäle variierten dabei zwischen 5000 und 25000 bzw. zwei und vier. Bei diesen Testmengen werden die CUDA-Kernels nur wenig beansprucht. Außerdem sind die Ergebnisse aus genannten Gründen teilweise ungenau. Die Anzahl von Samples und Audiokanälen kann durch Veränderung von Variablen vorgenommen werden. Diese müssen in den Modulen StreamWriter, AudioTracer und Bokeh Server verändert werden. Die Aufgaben der CUDA-Kernel werden generisch anhand dieser Parameter aufgeteilt, wodurch die Frage nach der Skalierbarkeit im Falle des AudioTracer-Moduls beantwortet werden kann. Ein sinnvoller nächster Schritt für die Erweiterung der Testreihen wären acht Audiokanäle mit einem Threshold von 160000 Samples. Um die maximale Skalierbarkeit bei größtmöglicher Genauigkeit zu testen ist es wichtig, dass die Erhöhung der Audiokanäle und Sample-Anzahl im besten Fall so gewählt wird, dass die Eingabemenge für der Fouriertransformation nach der Formel $\#channels * 20000$ gewählt wird. Die Genauigkeit würde dann bei 1Hz liegen. Die Grenzen dieses Moduls sind in der Theorie schwierig zu bestimmen, kann jedoch mit dem oben genannten Ansatz schnell bestimmt werden. Die Probleme werden vermutlich einen Bottleneck beim Empfangen der Daten über die TCP-Schnittstelle aufweisen. Hier könnten Verbesserungen erzielt werden. Im Kapitel 10 werden kurz die Vorteile der CUDA-Streams erwähnt, wodurch die Ausführung des Programms weiterläuft, während die CUDA-Aufgaben in einer Warteschlange eingereiht werden. Dadurch wäre das Warten auf die Verfügbarkeit der Daten weniger präsent.

Die Skalierbarkeit für das **AudioParcours** Teilprojekt im Vergleich schwieriger zu bestimmen. Das Projekt ist für die Manipulation einer Audioquelle ausgelegt. Eine Erweiterung des Quellcodes würde vorsehen, mehrere Audiostreams vom Paket Sounddevice zu erstellen. Sollte die Möglichkeit bestehen den jeweiligen Audiostream zu identifizieren, würde ein Callback reichen, um alle Streams in Abhängigkeit der Nutzereingaben zu manipulieren. Ansonsten würde der Quellcode aufgebläht, da für jeden Stream ein jeweiliger Callback vorhanden sein müsste. Außerdem muss im Voraus bekannt sein, an welcher Stelle sich die Audioquellen befinden. Die Klasse „Position“ müsste im Zusammenhang jedes Streams verwendet werden, da sich die Ausgangswerte unterscheiden. Dadurch wäre eine Untersuchung der Skalierbarkeit möglich. Ein anderer Ansatz wäre die Annahme, dass nicht die Anzahl der Audioquellen steigt, sondern die der Audiokanäle. Wären die Kanäle an festen Positionen „stationiert“, wäre es möglich, bei steigender Anzahl der Kanäle die Skalierung des Audioparcours zu untersuchen. Voraussichtlich würden jedoch ab einer gewissen Anzahl von Tonobjekten bzw. Audiokanälen eine Latenz zwischen Nutzereingaben und hörbarer Veränderung in der Wiedergabe entstehen.

9 Zusammenfassung

In dieser Arbeit wurden zwei Projekte vorgestellt, wobei das erstere AudioTracer Projekt größere Entwicklungs- und Recherchearbeit erforderte. Dieses besteht aus mehreren Modulen, die eigene Zuständigkeitsbereiche abdecken. Analog zu Abbildung 3 besteht dieses Teilprojekt aus den Modulen StreamWriter, jeweils einem CUDA Modul für C und Python und aus einem Bokeh Webserver Modul. Der StreamWriter liest eine WAV-Datei ein und macht die darauf gewonnenen Sampledaten für das C- oder Python-CUDA-Modul in Form eines TCP-Streams verfügbar. Das C CUDA-Modul empfängt diese Daten bis zu jeweils einem gewissen Threshold, wandelt diese in eine komplexe Repräsentation und startet eine CUDA-Routine. Jedem CUDA-Kernel-Thread wird ein Teilbereich der Daten zugewiesen. Die genaue Aufteilung kann der Sektion 5.3.2 entnommen werden. Jeder CUDA-Thread wendet auf die zugewiesenen Teilbereiche eine Fouriertransformation an. Im Falle des C-Moduls wird eine diskrete Fouriertransformation verwendet. Die Ausgabe beschreibt, welche harmonischen Frequenzen wie stark in den übermittelten Audiodaten anteilig sind.

Das Python CUDA-Modul weist einen ähnlichen Aufbau zum C-Modul auf. Prägnant ist jedoch, dass der Quellcode im Vergleich durch weniger notwendige Umformungen und Typtrue - wahrscheinlich mit Performance Verlusten - wesentlich kleiner ausfällt. Ein Performance Vergleich wurde hier nicht vorgenommen. Das Python Modul benutzt eine Fast Fourier Transformation aus dem Paket `cupy` um die Frequenzen zu extrahieren.

Beide CUDA-Module kommunizieren das Ergebnis Array über eine weitere TCP-Schnittstelle mit dem Python Bokeh Server Modul. Dabei handelt es sich um einen Visualisierungswebserver, der mithilfe von AJAX POST-Requests Daten an den Browser weiterleiten kann, die durch vorgefertigte JavaScript-Anbindungen in eine SVG-Grafik visualisiert werden.

Der Ablauf von StreamWriter bis zur Visualisierung im Browser findet im Kontext zu Grafik 3 von links nach rechts statt.

Das zweite Teilprojekt AudioParcours bietet eine Möglichkeit, eine WAV-Datei während der laufenden Wiedergabe über CUDA-Routinen zu manipulieren, um den Effekt einer positionsbezogenen Audiowiedergabe zu simulieren. Dieses Projekt wurde in Python implementiert. Python liefert die notwendigen Pakete `Sounddevice` und `PyCUDA`. Erstes ermöglicht eine Implementierung eines Audiostreams, wobei immer eine bestimmte Anzahl an Audiosamples wiedergegeben werden, bis eine Callback-Funktion aufgerufen wird, mit der die Audiodaten des nächsten Blocks festgelegt werden können. Parallel dazu können Nutzereingaben getätigt werden, die bewirken, dass CUDA-Operationen vorgenommen werden, die die Signaldaten in Abhängigkeit der getätigten Nutzereingaben manipuliert. Diese daraus gewonnen Daten werden in der Callback-Methode zu gegebenem Zeitpunkt verwendet.

10 Ausblick

Wie in der Einleitung erwähnt, gilt diese Arbeit als Versuch Basistechnologien zu präsentieren, mit denen eine CUDA-basierte positionsbezogene Schnittstelle für Audio entwickelt werden kann. Ein notwendiger Implementierungsschritt für dieses Ziel wäre die Verbindung der beiden Projekte AudioTracer und AudioParcours, sodass für die Soundmanipulation die anteiligen Frequenzen zu jedem Zeitpunkt bekannt sind. Mit diesem Schnittpunkt wäre es möglich, verschiedene Audiophänomene wie z.B. den Doppler-Effekt zu simulieren. Dafür müssten die anteiligen Frequenzen teilweise angepasst und mithilfe der inversen Fouriertransformation wieder in ein zusammengefügtes Tonbild konvertiert werden. Eine Möglichkeit für die Realisierung der hier präsentierten Technologien wäre es, spezielle Soundtreiber zu entwickeln, die direkt mit der CUDA-Schnittstelle der Grafikkarte kommunizieren können, was diesen Themenbereich jedoch nicht weniger komplex gestalten würde. Die Einleitung referenziert eine Hand voll Forschungsgebiete, in denen teure Hardware und Lokalitäten notwendig sind, um Audiophänomene zu untersuchen. Eventuell würde eine solche Schnittstelle eine „Digitalisierung“ in diesen Bereichen auslösen, oder zumindest im Zusammenspiel verwendet werden, um erforschte Soundphänomene digital abbilden zu können. Das AudioParcours Projekt weist noch Erweiterungsbedarf auf. Gegenwärtig wird das WAV-Format mit einer Unterstützung von bis zu zwei Audiokanälen erwartet. Die Formastreiktion könnte aufgehoben werden, um eine größere Breite von Audioformaten zu unterstützen. Das MP3-Format wäre ein Beispiel hierfür. Im AudioTracer Projekt wäre eine Überarbeitung der TCP-Streaming Logik sinnvoll. Bei den Testdurchläufen ist aufgefallen, dass die Daten vom StreamWriter zu langsam empfangen werden. Dadurch entsteht eine Verzögerung der CUDA-Routine. Unter anderem werden zusätzlich einige unnötige Konvertierungen zwischen Integer- und komplexen Darstellungen der Eingabefelder durchgeführt. Es wäre sinnvoll, diese initial in der komplexen Darstellungen vom StreamWriter Projekt auszulesen und auch in dieser Form zu versenden. Die CUDA-Routine des C-AudioTracer Projekts implementiert eine diskrete Fouriertransformation, bei der die jeweiligen Kernel-Threads einen nur kleinen Teilbereich der eigentlichen Samplingdaten zugewiesen bekommen. Eine individuelle Implementierung der Fast Fourier Transformation würde bei größeren Datensätzen Sinn ergeben.

Eine weitere Möglichkeit die Performance des vorliegenden Programms zu verbessern wäre die Benutzung von **CUDA-Streams**. Das Host-System würde in diesem Fall die auszuführenden Aufgaben in einer Warteschlangenstruktur am Device-System anmelden und direkt mit der Ausführung fortfahren, ohne auf die synchrone Ausgabe der CUDA-Routine zu warten. Dies würde den Zeitaufwand zwischen zwei Ausführungen der CUDA-Aufgaben im besten Fall komplett aufheben, sodass mehrere CUDA-Routinen ohne Aufschub ineinander überlaufen würden.

Die Allgemeine Komplexität der Initialisierung einer CUDA Kontexts in C wurde nicht gemessen, führt aber sehr wahrscheinlich dazu, dass mehr CPU Auslastung gebraucht wird als nötig wäre. Gegenwärtig wird nach einem Empfangen der Audio Samples jeweils

die Initialisierung des CUDA-Kontexts neu vorgenommen. Allgemein wäre interessant zu untersuchen, die parallele Berechnung der Audiofrequenzen schneller ist als eine sequentielle Fouriertransformation. Außerdem ist aus den Ergebnissen ersichtlich, dass die errechneten Frequenzen teilweise etwas ungenau sind. Wie bereits herausgestellt wurde, liegt dies an der Größe der Eingabedaten. Hier würde ein interessanter Aspekt sein, herauszufinden, ab welcher Größe der Eingabe die Ergebnisse genauer werden.

Abbildungsverzeichnis

1	Metainformationen des Audioformat WAV	5
2	Nyquist synthetischer Wind	6
3	Projektüberblick	13
4	fileConfig struct	14
5	Eingabe Datei im Binärmodus lesen	15
6	Vergleich fopen_s mit Modi „r“ (oben) und „rb“ (unten)	15
7	Hauptiteration: Extraktion der Sample Daten	17
8	StreamWriter Main Loop	18
9	Sequenzdiagramm StreamWriter	19
10	Socketinitialisierung	20
11	Schleifenoperation: Hinterer Teil	22
12	Schleifenoperation: Vorderer Teil	24
13	Sequenzdiagramm AudioTracer-C	26
14	CUDA Vorbereitungen	28
15	Topologie der CUDA Kernels [14]	29
16	Gridbeispiel	30
17	CUDA Kernel	30
18	Kernelfunktion _dft	32
19	Erstellung eines Sockets in Python	33
20	Ausführung CUDA FFT in Python	34
21	Sequenzdiagramm AudioTracer-Python	35
22	Erstellung eines Plots mit Ajax Datenübertragung	36
23	JavaScript-Code zur Erstellung der Datenarrays	38
24	Test 1	40
25	Test 3	40
26	Test 5	41
27	Test 7	41
28	Test 2	42
29	Test 4	42
30	Test 6	43
31	Test 8	43
32	Überblick Teilprojekt Audioparcours mit CUDA	45
33	Positionsveränderung Audioparcours	46
34	Sequenzdiagramm Audioparcours	48
35	Verwendete Pakete im Projekt Audioparcours	49
36	Python: dataclass	49
37	Audioparcours: Einlesen der WAV-Datei	50
38	Audioparcours: Erstellung des CUDA-Kontext	50
39	Audioparcours: Globale Variablen	51
40	Audioparcours: SourceModule	52
41	Audioparcours: Sounddevice Stream	54

42	Audioparcours: Sounddevice Callback-Methode	54
43	Audioparcours: Nutzereingaben	56
44	Audioparcours: Manipulation über CUDA	57

Literatur

- [1] R. Aachen. „Forschungsgebiete Hörtechnik und Akustik der RWTH Aachen.“ (), Adresse: <https://www.akustik.rwth-aachen.de/cms/Institut-fuer-Hoertechnik-und-Akustik/Forschung/~fnru/Forschungsgebiete-Medizinische-Akustik/> (besucht am 04.07.2021).
- [2] btarunr. „Tech Powerup Artikel zu Audio Raycasting.“ (), Adresse: <https://www.techpowerup.com/246820/nvidia-does-a-trueaudio-rt-cores-also-compute-sound-ray-tracing> (besucht am 04.07.2021).
- [3] F. Hasselbach. „Quellcode der Abschlussarbeit.“ (), Adresse: https://github.com/fahaba/Master_Thesis (besucht am 04.07.2021).
- [4] B. L. D. P. Team. „WAVE PCM soundfile format.“ (2021), Adresse: https://wiki.dpconline.org/images/archive/4/46/20160217163258!WAV_Assessment_v1.0.pdf (besucht am 04.07.2021).
- [5] C. S. soundfile++ App (Sapp). „WAV Format Assessment.“ (2021), Adresse: <http://soundfile.sapp.org/doc/WaveFormat/> (besucht am 04.07.2021).
- [6] Audacity. „Nyquist Audacity.“ (2021), Adresse: <https://www.audacityteam.org/about/nyquist/> (besucht am 04.07.2021).
- [7] 'Alex'. „Wind Beispiel Audacity Forum.“ (2021), Adresse: <https://www.audacity-forum.de/download/edgar/nyquist/nyquist-doc/examples/rbd/15-wind-tutorial.htm> (besucht am 04.07.2021).
- [8] R. B. Dannenberg. „Nyquist Reference Manual.“ (2021), Adresse: <http://www.cs.cmu.edu/~rbd/doc/nyquist/nyquistman.pdf> (besucht am 04.07.2021).
- [9] P. Arenz. „Arenz Fouriertransformation.“ (2005), Adresse: <https://www.math.uni-trier.de/~schulz/Prosem-0405/Arenz.pdf> (besucht am 04.07.2021).
- [10] C. Scheider. „De Moivre Theorem.“ (2011), Adresse: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.722.2485&rep=rep1&type=pdf> (besucht am 04.07.2021).
- [11] N. Corporation. „Offizielle CUDA Webseite NVIDIA Developer.“ (2021), Adresse: <https://developer.nvidia.com/cuda-zone> (besucht am 29.06.2021).
- [12] —, „Nvidia Geforce GTX 10 Serie.“ (2021), Adresse: <https://www.nvidia.com/de-de/geforce/10-series/> (besucht am 29.06.2021).
- [13] Microsoft. „Microsoft Docs für socket.“ (), Adresse: <https://docs.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-socket> (besucht am 04.07.2021).
- [14] NVIDIA. „CUDA Developer Blog.“ (), Adresse: <https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/> (besucht am 04.07.2021).
- [15] —, „CUDA docs.“ (), Adresse: <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html> (besucht am 04.07.2021).

- [16] P. Dokumentation. „PyCUDA Docs.“ (), Adresse: <https://document.tician.de/pycuda/> (besucht am 04.07.2021).

11 Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich diese Masterarbeit eigenständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Arbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher auch nicht veröffentlicht.

Windeck, den 11. November 2021

Unterschrift: _____