Heterogeneous Computing

Portfolio

Aufgabensammlung

Universität Trier
FB IV - Informatikwissenschaften
Professur für Systemsoftware und Verteilte Systeme

Semester: SoSe 2024

Name und Matrikelnummer:

Simon Szulik, 1474315

Inhaltsverzeichnis

1. Fourieranalyse der geheimnisvollen Wellenlängen

Die Fourieranalyse ist eine mathematische Methode, die es ermöglicht, komplexe periodische Funktionen in eine Reihe von einfachen sinusförmigen Wellen zu zerlegen. Sie wird häufig in der Signalverarbeitung und der Analyse periodischer Phänomene wie Schwingungen und Wellen angewendet, um deren Bestandteile und Frequenzanteile zu untersuchen. Durch die Anwendung der Fourieranalyse kann man komplexe Signale in ihre Grundbestandteile zerlegen und dadurch besser verstehen.

1.1 Implementierung

Für die Implementierung wurde Python verwendet, sowie die folgenden Bibliotheken:

- **scipy**: Zum Einlesen der Audio-Datei (.wav) & dem Anwendungen einer schnellen Fourier-Analyse.
- **numpy**: Für numerische Operationen und die Berechnung der einzelnen Frequenzen.
- **matplotlib**: Als visuelle Darstellungsmöglichkeit der Daten und zum Erstellen von Plots.
- **argparse**: Zum Parsen der Kommandozeilenargumente, um die Eingabeparameter wie Dateiname, Blockgröße und Ausgabetyp zu bestimmen.
- **memory profiler**: Zur Analyse und Auswertung des Speicherbedarfs.

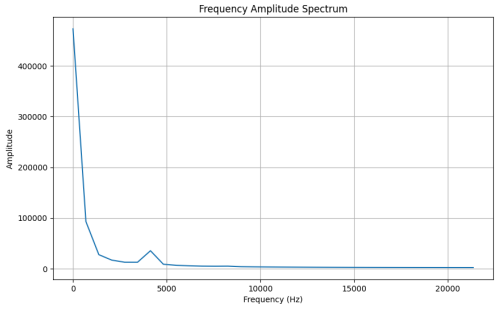
Im Beispiel 2.1 ist eine typische Ausführung des Skriptes zu sehen. Mit hilfe der Argumente lässt sich sowohl die Blockgröße, als auch die Art des Diagramms einfach bestimmen.

```
1  py.exe Fourier.py audio.wav —b 512 —o f —p 1
```

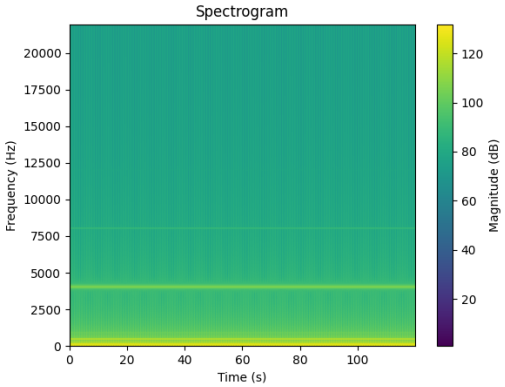
Beispiel 1.1: Skriptausführung in der Knsole

1.1.1 Ergebnisse

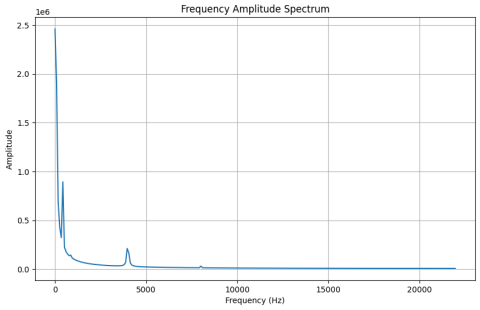
In diesem Abschnitt betrachten wir ein paar Beispiel Plots der Analyse. Bezüglich der Blockgröße gibt es einige Charakteristika die zu beachten sind. Behandelt man Signale mit schnellen und häufigen Änderungen wie Sprachaufnahmen, so können kleinere Blöcke (128-512) besser sein. Signale mit stabileren Frequenzen können Problemlos mit größeren Blocken analysiert werden. Um eine möglichst breite Spanne von Fällen abzudecken, befindet sich in Abbildung 1.2 eine Ansammlung von verschiedenen Plots der zu analysierenden Datei.



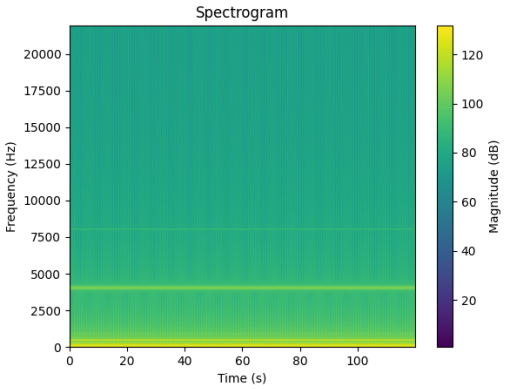
(a) Frequenz-Plot mit Blockgröße 64



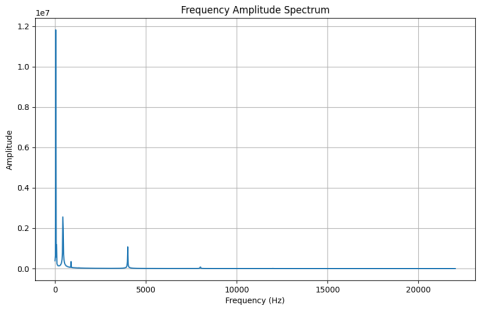
(b) Spektrogramm mit Blockgröße 64



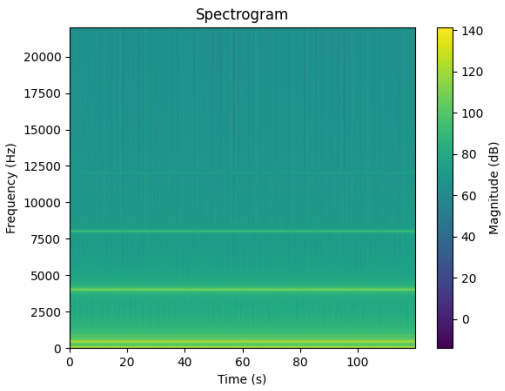
(c) Frequenz-Plot mit Blockgröße 512



(d) Spektrogramm mit Blockgröße 512



(e) Frequenz-Plot mit Blockgröße 2048



(f) Spektrogramm mit Blockgröße 2048

Abbildung 1.1: Beispielergebnisse

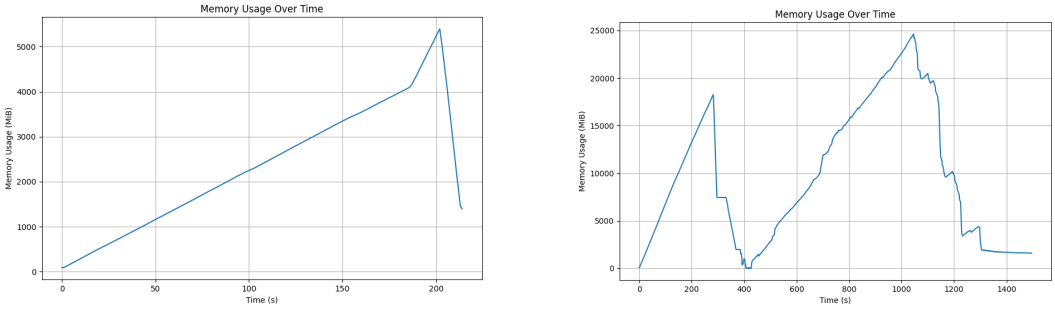
1.2 Speicherplatzbedarf

Für eine erste Speicheranalyse wurden die folgenden Spezifikationen unter einem Windows System verwendet:

Attribut	**Details**
Prozessor	AMD Ryzen 5 7500F
Architektur	ZEN 4
Kerne	6
Threads	12
Basis-Taktfrequenz	4.8 GHz
L3-Cache	32 MB
Fertigungstechnologie	65 Watt
Speicherunterstützung	DDR5
PCIe-Unterstützung	PCIe 5.0
Sockel	AM5

Tabelle 1.1: Spezifikationen des AMD Ryzen 5 7500F

Wie erwartet und in Abbildung 1.2 zu erkennen, erhöht sich mit steigender Blockgröße auch der benötigte RAM-Verbrauch. Dies geschieht aufgrund der immer größer werdenden Vektoren, die benötigt werden, um die Daten zu speichern. Bei 32 GB verfügbarem RAM ist eine Blockgröße von 512 das Maximum, bevor der Prozess aufgrund des Speicherplatzbedarfs abbricht.



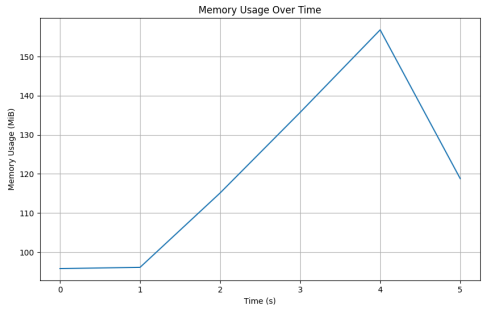
(a) Speicherbedarf mit Blockgröße 64 (b) Speicherbedarf mit Blockgröße 512

Abbildung 1.2: Speicherplatzbedarf mit Schrittweite 1

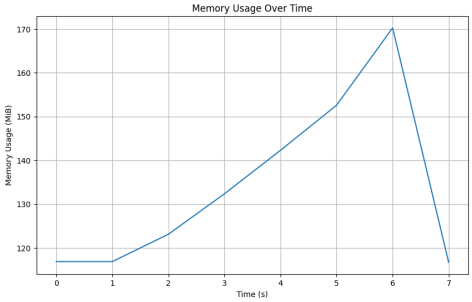
Um dennoch Zugriff zu größeren Blockgrößen und deren Speicherverbauch zu haben, gibt es einige Lösungsansätze. Eine effektive Methode zur Reduktion des Speicherverbrauchs besteht darin, die Präzision der Daten zu verringern. Audiodaten werden oft in 64-Bit-Gleitkommazahlen (float64) gespeichert, die jedoch mehr Speicherplatz beanspruchen als notwendig. Wenn die Präzision ausreicht, kann die Konvertierung der Daten in 32-Bit-Gleitkommazahlen (float32) den Speicherbedarf halbieren. Dies kann durch einfache Typumwandlung erreicht werden und führt zu einer signifikanten Reduktion des Speicherverbrauchs, ohne die Qualität der Analyse erheblich zu beeinträchtigen.
Ein weiterer Ansatz zur Reduktion des Speicherverbrauchs besteht darin, nur die notwendigen Informationen zu speichern. Anstatt alle FFT-Blöcke zu speichern, können beispielsweise nur die durchschnittlichen Magnituden oder die Spitzenwerte der

Frequenzen aufgezeichnet werden. Dies reduziert die Menge der gespeicherten Daten erheblich, da nur aggregierte oder besonders wichtige Informationen behalten werden. Diese Methode eignet sich besonders, wenn die detaillierten Daten der einzelnen Blöcke nicht benötigt werden.
Eine erweiterte Methode zur FFT-Berechnung besteht darin, die Analyse in einem sliding Window durchzuführen und die Ergebnisse zwischendurch in temporären Dateien zu speichern. Diese Chunk-basierte Verarbeitung, teilt die Audiodaten in kleinere Abschnitte auf, die in den verfügbaren RAM passen. Nach der Verarbeitung eines Chunks werden die Ergebnisse gespeichert und der nächste Chunk wird geladen. Diese kontinuierliche Verarbeitung verhindert, dass der gesamte Speicher belegt wird und ermöglicht dennoch eine detaillierte Analyse der Audiodaten.
Da es sich bei der analysierten WAV-File allerdings um keine Sprachaufnahme, sondern um einen kontinuirlichen Frequenzmix handelt, ist eine feine zeitliche Auflösung nicht unbedingt erforderlich. Ein sinvoller Lösungsansatz um den Speicherverbrauch zu deutlich zu reduzieren wäre somit eine größere Schrittweite. Anstatt die Blöcke um jeweils nur einen Schritt zu verschieben, können sie um mehrere Schritte verschoben werden. Dies reduziert die Anzahl der berechneten Blöcke und somit auch die Menge der gespeicherten Daten.

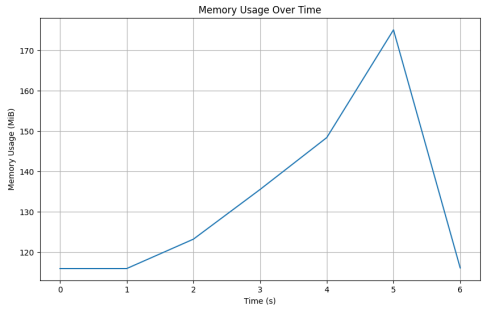
Ein simpler Ansatz die Schrittweite zu varrieren wäre es, sie gleich der angewendeten Blockgröße zu setzen. Das Ergebniss wären hierbei, dass ausschließlich sich nicht überlappende Blöcke betrachtet werden. Abbildung 2.5 beinhaltet eine Speicheranalyse mit genau diesen Einstellungen. Leicht zu erkennen ist, das sowohl der benötigte Speicherplatz und die Berechnungszeit sich um ein vielfaches verkleinert hat. Dadurch können wir weitaus größere Blockgrößen betrachten.



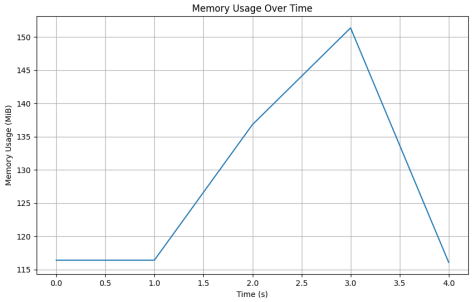
(a) Speicherbedarf mit Blockgröße 64



(b) Speicherbedarf mit Blockgröße 512



(c) Speicherbedarf mit Blockgröße 1024



(d) Speicherbedarf mit Blockgröße 2048

Abbildung 1.3: Speicherplatzbedarf mit Blockgroßer Schrittweite

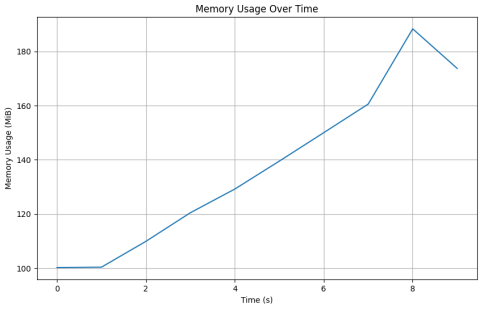
1.3 Variationen

Als Variation, wurde ein andere Prozessor verwendet, um die Unterschiede zwischen einem AM und BGA Sockeln festzustellen, als auch unterschiedliche Betriebssysteme.

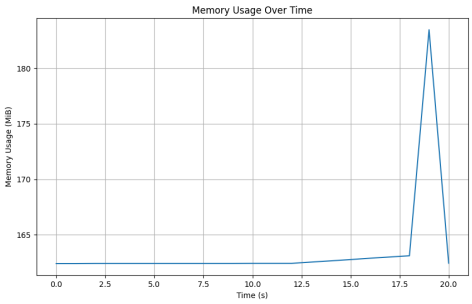
Attribut	**Details**
Prozessor	Intel Core i5-8250U
Architektur	Kaby Lake R
Kerne	4
Threads	8
Basis-Taktfrequenz	1.6 GHz
L3-Cache	6 MB
Fertigungstechnologie	14nm
Speicherunterstützung	DDR4-2400, LPDDR3-2133
PCIe-Unterstützung	PCIe 3.0
Sockel	BGA 1356

Tabelle 1.2: Spezifikationen des Intel Core i5-8250U

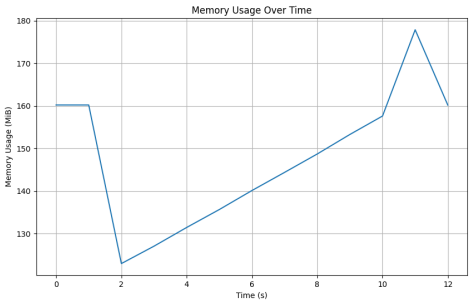
Auf dem oben beschriebenen Prozessor war sowohl unter Linux als auch unter Windows bereits bei kleinsten Blockgrößen ein Engpass festzustellen. Die Plots in Abbildung 2.4 zeigen die Ergebnisse der Speicheranalyse unter Linux. Ähnlich zum Prozessor aus den vorherigen Aufgaben, benötigen wir hier nur auch für größere Blöcke aufgrund der Blockgroßen Schrittweite nur wenige Sekunden für die Berechnung.



(a) Speicherbedarf mit Blockgröße 64



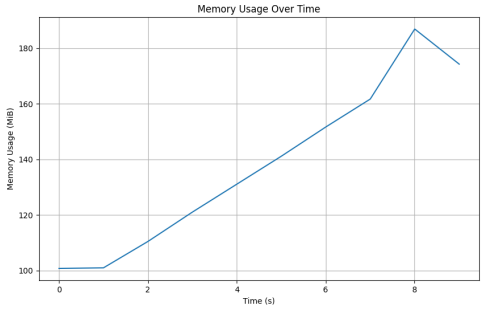
(b) Speicherbedarf mit Blockgröße 512



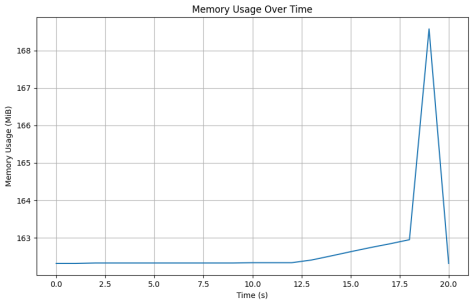
(c) Speicherbedarf mit Blockgröße 2048

Abbildung 1.4: Speicherplatzbedarf mit Blockgroßer Schrittweite unter Linux

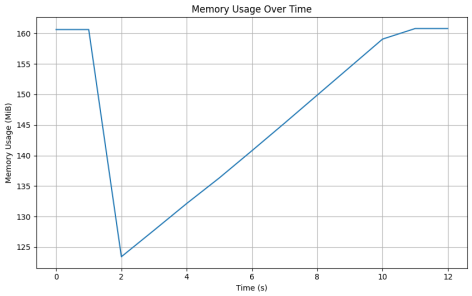
Führen wir die gleichen Tests auf gleichem Prozessor unter Windows durch, so scheint die Anwendung unter Windows etwas weniger Speicher zu verbrauchen. Diese Unterschiede können auf eine Vielzahl von Faktoren zurückzuführen sein, die sowohl in den Betriebssystemen als auch in deren spezifischer Implementierung und Verwaltung von Ressourcen liegen. Windows und Linux haben jeweils ihre eigenen Methoden zur Verwaltung von Speicher, die unterschiedliche Auswirkungen auf die Effizienz der Speichernutzung haben können. Windows verwendet ein Speichermanagementsystem, das häufig auf die Bedürfnisse von Desktop-Anwendungen optimiert ist. Es nutzt Techniken wie den "Heap Manager", der Speicher effizient verwaltet und Fragmentierung minimiert. Windows tendiert dazu, aggressiver ungenutzten Speicher zurückzufordern und Prozesse priorisiert zu behandeln, die eine hohe Leistung erfordern.



(a) Speicherbedarf mit Blockgröße 64



(b) Speicherbedarf mit Blockgröße 512



(c) Speicherbedarf mit Blockgröße 2048

Abbildung 1.5: Speicherplatzbedarf mit Blockgroßer Schrittweite unter Windows

2. CPU vs. GPU

Im folgenden Kapitel werden verschiedene Implementierungen der Fourier-Analyse gegenübergestellt. Dabei liegt der Fokus auf der Bewertung sequenzieller Lösungen sowie paralleler Ansätze, die sowohl CPU als auch GPU-Kerne nutzen. Die sequenzielle Implementierung nutzt traditionell einen einzelnen Prozessorkern, um die Fourier-Transformation schrittweise auf einem Datensatz auszuführen. Diese Methode bietet eine einfache und zuverlässige Möglichkeit, die Frequenzcharakteristiken eines Signals zu analysieren, ist jedoch durch die Begrenzung auf die Rechenleistung eines einzelnen Kerns limitiert.
Im Gegensatz dazu zielen parallele Ansätze darauf ab, die Verarbeitungslast über mehrere Kerne oder sogar über spezialisierte Prozessoren wie GPUs zu verteilen. Durch diese Parallelisierung kann die Analysegeschwindigkeit erheblich gesteigert werden, insbesondere bei großen Datensätzen. Als Datensätze werden sowohl die bereitgestellten Dateien als auch eigens erstellte WAV-Testdateien verwendet.

Die Spezifikationen der CPU bzw. die GPU, mit der die Daten berechnet wurden, sind in den beiden folgenden Tabellen festgehalten.

Attribut	**Details**
Prozessor	AMD Ryzen 5 7500F
Architektur	ZEN 4
Kerne	6
Threads	12
Basis-Taktfrequenz	4.8 GHz
L3-Cache	32 MB
Fertigungstechnologie	7 nm
Speicherunterstützung	DDR5
PCIe-Unterstützung	PCIe 5.0
Sockel	AM5

Tabelle 2.1: Spezifikationen des AMD Ryzen 5 7500F

Attribut	**Details**
Grafikprozessor	Radeon RX 7800 XT
Architektur	RDNA 3
Compute Units	80
Stream-Prozessoren	5120
Basis-Taktfrequenz	2.4 GHz
Boost-Taktfrequenz	2.8 GHz
VRAM	16 GB GDDR6
Speicherbandbreite	512 GB/s
Stromverbrauch	250 Watt
PCIe-Unterstützung	PCIe 4.0

Tabelle 2.2: Spezifikationen der AMD Radeon RX 7800 XT

2.1 Erstellen eigener WAV-Testdateien

In dem Skript „Test_Wave_Generator “werden insgesamt 4 WAV-Dateien erstellt. Es werden Sinuswellen mit 440 Hz für 10 und 100 Sekunden und Stille für 300 Sekunden erstellt. Jede Funktion erzeugt das entsprechende Audiosignal, indem sie Werte für die angegebenen Dauer und den Abtastratenbereich berechnet. Die generierten Audiodaten werden dann in WAV-Dateien gespeichert. Die Funktion save_wave konvertiert die Audiodaten in das erforderliche Format und speichert sie in Dateien ab.

2.2 Sequenzielle Implementierung in Python

Die sequenzielle Lösung lehnt sich stark an die in Übung 1 in Python implementierte Version an. Die Hauptänderung besteht darin, die Schrittweite ebenfalls in den Programmparametern übergeben werden kann. Außerdem gibt es einen eigenen Parameter, der den Schwellwert für den Amplitudenmittelwert bestimmt.

Im Beispiel 2.1 ist eine typische Ausführung des Skriptes aufgeführt.

```
1  py.exe Fourier_Seq.py audio.wav —b 512 —o 1 —t 10000
```

Beispiel 2.1: Skriptausführung in der Knsole

2.3 Parallele Implementierung in Python

Die multiprocessing-Bibliothek spielt eine zentrale Rolle bei der Parallelisierung der FFT-Berechnungen. Durch die Verwendung von Pool und starmap wird der Code so strukturiert, dass die Berechnung der FFT für jeden Block auf separaten Prozessorkernen erfolgt. Dadurch wird die Verarbeitungsgeschwindigkeit insbesondere bei großen Audiodateien erheblich verbessert, da die Berechnungslast effizient auf mehrere CPU-Kerne verteilt wird

Die parallele Verarbeitung wird in der Funktion `perform_fft` verwendet, um die FFT für mehrere Datenblöcke gleichzeitig auszuführen. Dies geschieht in den folgenden Schritten:

- **Datenblockaufteilung**: Die gesamten Daten werden in Blöcke unterteilt, wobei ein `offset` zwischen aufeinanderfolgenden Blöcken definiert ist.
- **Pool Erstellung**: Ein Pool von Prozessen wird erstellt, der so viele Prozesse enthält wie CPU-Kerne (`cpu_count()`).
- **Parallelisierung**: Mit `pool.starmap` wird die Funktion `perform_fft_block` auf jedes Element (`block`) der Liste `blocks` angewendet. Dies führt dazu, dass die FFT-Berechnungen für jeden Block parallel auf verschiedenen Prozessorkernen ausgeführt werden.
- **Rückgabe der Ergebnisse**: Die Ergebnisse der FFT für jeden Block werden gesammelt und als Liste von Arrays (`fft_blocks`) zurückgegeben.

2.4 Parallele Implementierung mit OpenCL

Die Implementierung der FFT (Fast Fourier Transform) auf der GPU unter Verwendung von OpenCL umfasst mehrere wichtige Schritte, von der Einrichtung des OpenCL-Kontexts bis zur Ausführung des FFT-Kernels auf der GPU. Bevor wir zum eigentlichen Code kommen, müssen wir die OpenCL-Umgebung einrichten, was die Auswahl der Plattform, das Erstellen eines Kontexts und dem Erstellen einer Befehlswarteschlange entspricht.

Algorithm 1 OpenCL-Kontext und Befehlswarteschlange erstellen

[Erstellen der AMD Plattform und des Kontexts]

```
platform ← cl.get_platforms()[0]
context ← cl.Context(dev_type=cl.device_type.GPU, platform)
queue ← cl.CommandQueue(context)
```

Der Kern der GPU-Implementierung ist der OpenCL-Kernel. Dieser Kernel läuft auf der GPU und führt die FFT-Berechnungen durch.

Algorithm 2 FFT-Kernel in OpenCL

function FFT($data, fft\_blocks, block\_size, offset, num\_blocks$)
  for $i \leftarrow 0$ **to** $num\_blocks - 1$ **do**
    for $j \leftarrow 0$ **to** $block\_size/2 - 1$ **do**
      $real \leftarrow 0.0$
      $imag \leftarrow 0.0$
      for $k \leftarrow 0$ **to** $block\_size - 1$ **do**
        $angle \leftarrow -2.0 \times 3.141592653589793 \times j \times k/block\_size$
        $real \leftarrow real + data[i \times offset + k] \times \cos(angle)$
        $imag \leftarrow imag + data[i \times offset + k] \times \sin(angle)$
      end for
      $fft\_blocks[i \times (block\_size/2) + j] \leftarrow \sqrt{real \times real + imag \times imag}$
    end for
  end for
end function

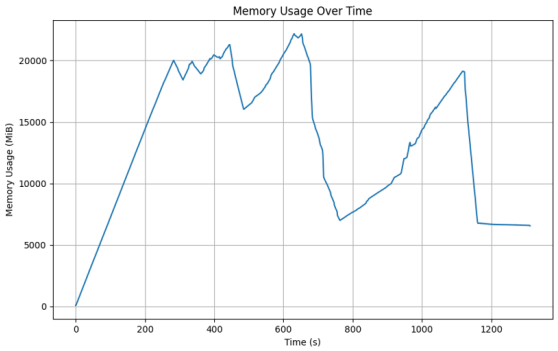
Der Kernel-Code definiert die Operationen, die jeder Thread (Work-Item) auf der GPU ausführt. In diesem Fall führt der Kernel die FFT-Berechnung für jeden Block von Audiodaten aus. Jedes Work-Item verarbeitet einen Block von Daten, führt eine Schleife durch, um die FFT für jede Frequenz zu berechnen, und speichert die Ergebnisse in einem Ausgabe-Array. Die Berechnung erfolgt unter Verwendung von trigonometrischen Funktionen für die Real- und Imaginärteile der FFT.
Nachdem der Kernel-Code geschrieben wurde, wird er mit cl.Program(context, program_src).build() kompiliert. Eventuelle Kompilierfehler werden abgefangen und ausgegeben. Danach werden Speicherbereiche (Buffers) für die Eingabe- (data_buffer) und Ausgabedaten (fft_buffer) erstellt. Der Kernel wird dann mit program.fft(queue, (num_blocks,), None, data_buffer, fft_buffer, np.int32(block_size), np.int32(offset), np.int32(num_blocks)) aufgerufen. Hierbei werden die notwendigen Argumente an den Kernel übergeben, einschließlich der Puffer für die Daten und Parameter wie Blockgröße und Anzahl der Blöcke. Nach der Ausführung des Kernels werden die Ergebnisse von fft_buffer zurück in das fft_blocks-Array auf dem Host kopiert, um sie weiter zu verarbeiten oder anzuzeigen.

2.5 Ergebnisse und Vergleich

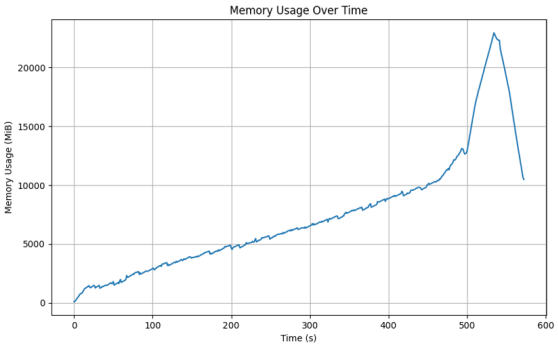
2.5.1 CPU Varianten

Angefangen mit der bereits in der ersten Übung ausgiebig getesteten Datei **nicht_zu_laut_abspielen.wav**, lässt sich ein Speedup von 100% feststellen. Dabei wurde die Testdatei mit einer Blockgröße von 512 und einem Versatz von 1 berechnet.

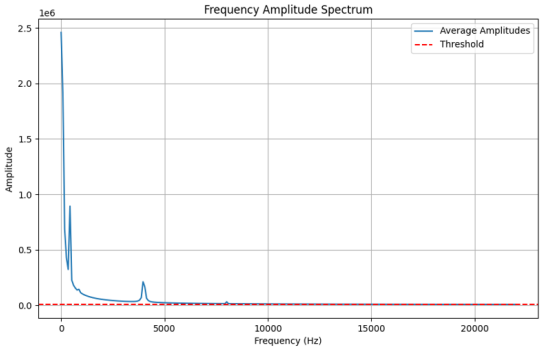
Während die Sequenzielle Python Lösung ganze 12 Minuten zur Berechnung benötigt, benötigt die parallele Variante nur die hälfte der Zeit. Der maximale RAM-Verbrauch verändert sich jedoch nur geringfügig.



(a) Sequenzielle Lösung



(b) Parallele Lösung

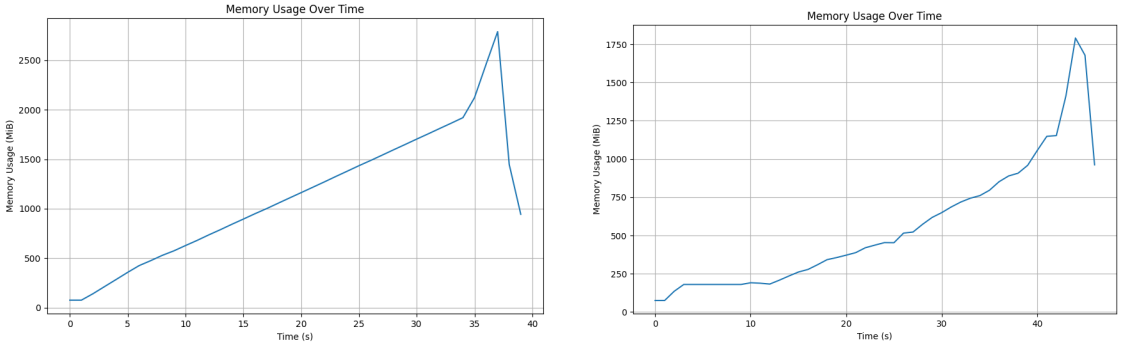


(c) Frequenzdiagramm mit Schwellenwert von 10000 für den Amplitudenwert

Abbildung 2.1: Berechnung mit Blockgröße 512 und Versatz 1

Beim Testen der kürzesten Datei **test_440hz_10s.wav** ist wie erwartet die Sequenzielle CPU Lösung schneller als die Parallele. Kurze Dateien bedeuten typischerweise weniger Daten, die verarbeitet werden müssen, und dadurch verringert sich der potenzielle Nutzen der Parallelisierung. Bei paralleler Verarbeitung entsteht ein zusätzlicher Overhead durch das Starten und Synchronisieren von Prozessen oder Threads. Dieser Overhead kann bei kurzen Dateien die Vorteile der Parallelisierung überwiegen und zu langsameren Gesamtlaufzeiten führen.
Ein weiterer Faktor sind die Kommunikationskosten zwischen den Prozessen oder Threads. Bei paralleler Verarbeitung müssen Daten zwischen den verschiedenen Einheiten ausgetauscht werden, was zusätzliche Zeit und Ressourcen beansprucht. Bei kurzen Dateien, wo die Verarbeitungszeit an sich schon gering ist, können diese Kommunikationskosten ins Gewicht fallen und die Geschwindigkeit beeinträchtigen. Zudem könnten bei paralleler Verarbeitung nicht alle CPU-Kerne effizient ausgelastet werden, insbesondere wenn die Aufteilung der Arbeit zwischen den Kernen

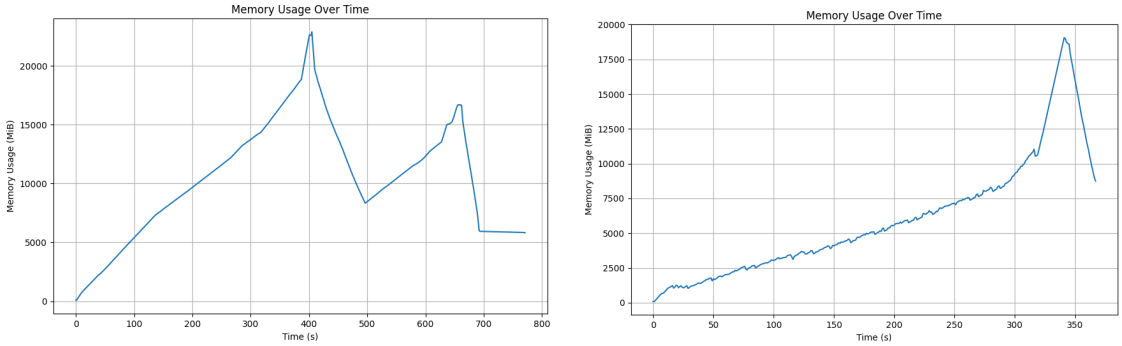
ungleichmäßig ist oder wenn die Verarbeitungsaufgabe so klein ist, dass die Ressourcenausnutzung ineffizient wird. Auffällig ist jedoch, dass die sequenzielle Lösung deutlich mehr Speicher verbraucht.



(a) Sequenzielle Lösung (b) Parallele Lösung

Abbildung 2.2: Berechnung mit Blockgröße 512 und Versatz 1

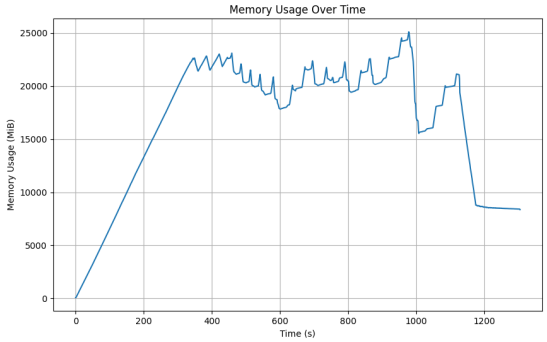
Bei der fft macht die Art der untersuchten Wellenlängen und Frequenzen keinen unterschied. Die Länge der Datei spielt jedoch eine große Rolle. Bei gleichbleibender Frequenz von 440hz, jedoch dem 10-fachen an Länge zeigt sich ein anderes Ergebniss. Hier erreichen wir einen Speedup von knapp 100% wobei die parallele Lösung erneut weniger Speicher benötigt.



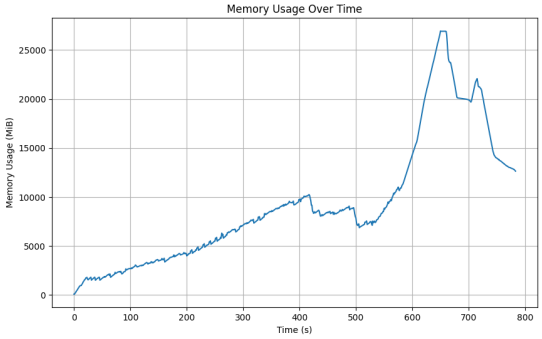
(a) Sequenzielle Lösung (b) Parallele Lösung

Abbildung 2.3: Berechnung mit Blockgröße 512 und Versatz 1

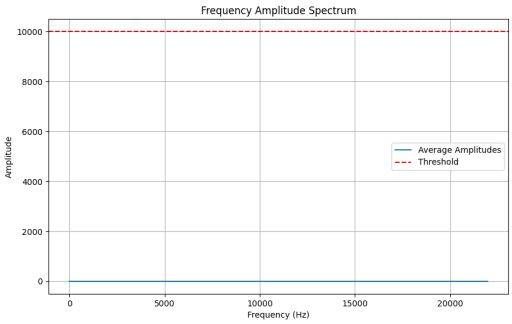
Noch genauer zu erkennen ist der eigentliche Speedup, wenn man die analysierten Frequenzen komplett ignoriert. Die nachfolgenden Plots enthalten die Ergebnisse der fft einer WAV-Datei mit 150s Stille. Diese Tabellen dieser Audiodatei sind komplett mit Nullen gefüllt. Der erzielte Speedup bei diesen Tests ist knapp 50%.



(a) Sequenzielle Lösung



(b) Parallele Lösung

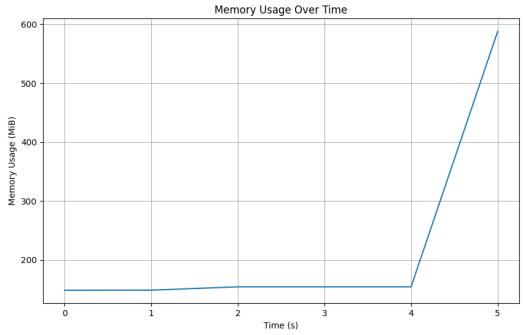


(c) Frequenzdiagramm mit Schwellenwert von 10000 für den Amplitudenwert

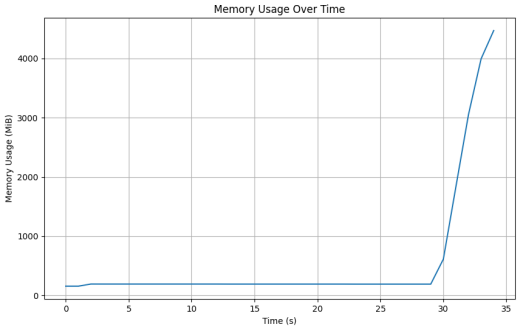
Abbildung 2.4: Berechnung mit Blockgröße 512 und Versatz 1

2.5.2 GPU Variante

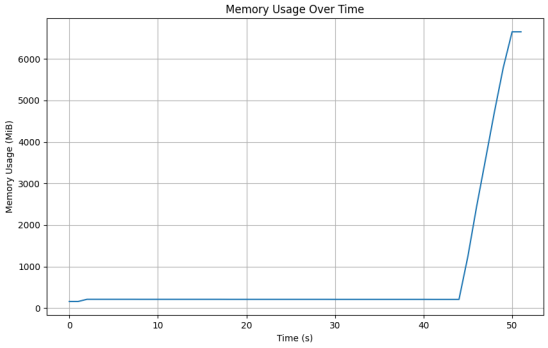
Wie erwartet bietet die Berechnung auf der GPU einen deutlichen Speedup. Sowohl die sequenzielle Lösung als auch Parallelität auf der CPU sind noch um eingies langsamer.



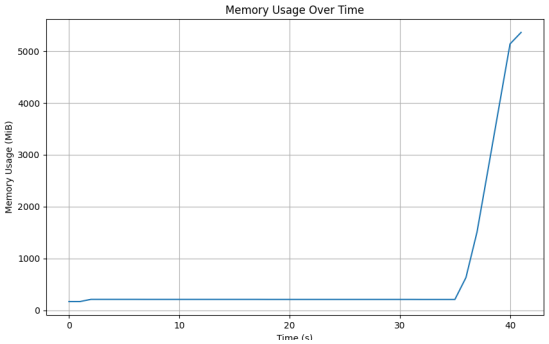
(a) 440Hz auf 10s



(b) 440Hz auf 100s



(c) Stille auf 150s



(d) nicht_zu_laut_abspielen

Abbildung 2.5: Berechnung mit Blockgröße 512 und Versatz 1

2.6 Gesamtvergleich

Vergleichen wir nun alle implementierten Versionen, so fällt auf, dass die OpenCL-Version um ein Vielfaches schneller ist. Die Single-Core-CPU-Lösung schneidet bei sehr kleinen Dateien noch gut ab, wird aber trotz Overhead schnell von der Multi-Core-CPU-Lösung überholt. Dabei erreichen wir einen Speedup von bis zu 150% bei größeren WAV-Dateien. Die GPU-Implementierung benötigt für die größte WAV-Datei gerade einmal 50 Sekunden, während selbst der Multi-Core-CPU-Ansatz noch einige Minuten benötigt. Bei der 10 Sekunden langen Testdatei erhalten wir ebenfalls eine gravierende Verbesserung von 40 bis 50 Sekunden auf knapp 5 Sekunden.

Dies ist auf die GPU-Architektur zurückzuführen, die insbesondere mathematische Operationen wie die FFT sehr effizient und parallel ausführen kann, indem Anweisungen gleichzeitig auf mehrere Datenpunkte angewandt werden.

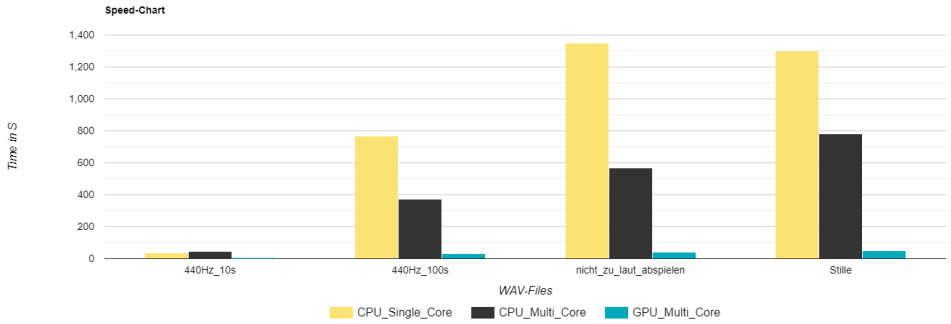


Abbildung 2.6: Vergleich der verschiedenen Implementierungen