Heterogeneous Computing
Übungsblatt 1

Kilian Bartz (Mknr.: 1538561)

Hinweis. Der zugehörige Code befindet sich im folgenden Github Repository: `https://github.com/kilianbartz/hetcomp`.

Aufgabe 1

Für die Lösung dieser Aufgabe habe ich mich für die Numpy Implementierung[1] der *Fast Fourier Analysis* entschieden. Die Audiodatei (dazu wurde immer "nicht_zu_laut_abspielen.wav"verwendet) wird mithilfe von scipy[2] eingelesen. Da es sich um eine Stereodatei handelt, verwerfe ich den 2. Audiokanal und verwende ausschließlich den ersten. Die Fourieranalyse wird auf Blöcken der Größe 2205 (50ms) durchgeführt. Für jeden Block werden die Hauptfrequenzen (> 50dB; willkürlich gewählter Schwellwert) gespeichert und später in eine CSV-Datei geschrieben. Plottet man diese für den ersten Block, ergeben sich die Frequenzen, die in Abbildung 1 zu sehen sind.

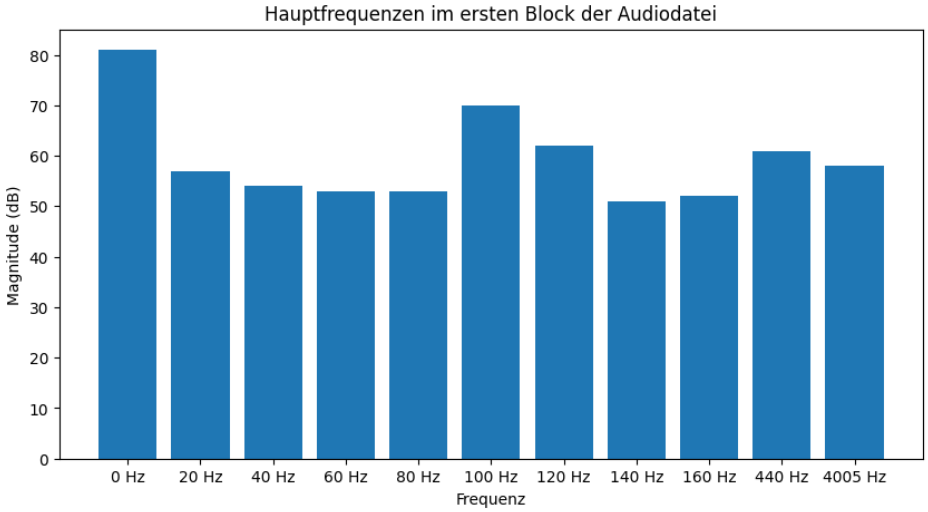


Abbildung 1: Hauptfrequenzen und ihre Amplituden für die Datei "nicht_zu_laut_abspielen.wav"

Aufgabe 2 und 3

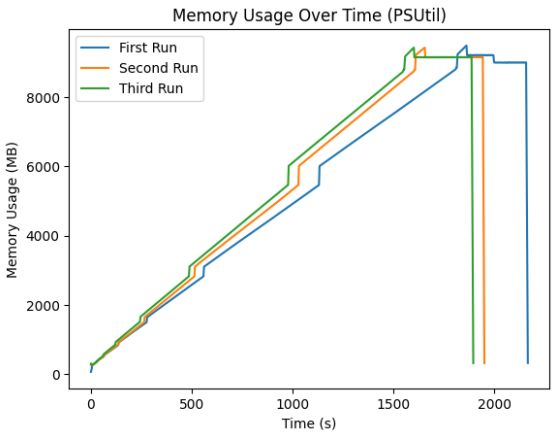
Um die unterschiedlichen Speicherverbräuche der Fourier-Analyse auf verschiedenen Plattformen zu untersuchen, wurde der Speicherverbrauch auf drei Plattformen gemessen:

1. Windows 11 Rechner mit 32 GB RAM, AMD 7700X (x86 CPU)
2. Ubuntu Server mit 128 GB RAM, Intel Core i7 8700 (x86 CPU)
3. Raspberry Pi 5 (Pi OS) mit 8 GB RAM, Broadcom BCM2712 (ARM Cortex A76 CPU).

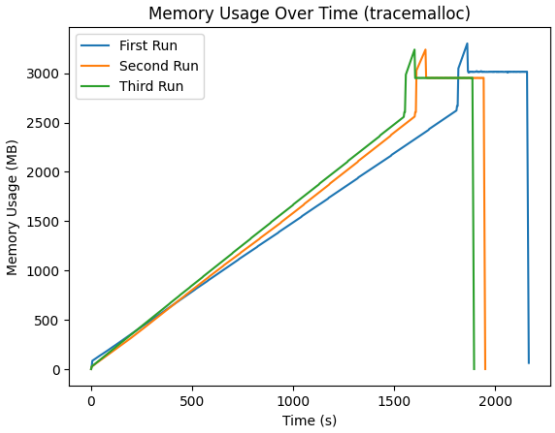
Diese Testsysteme erlauben es, betriebsystemabhängige Unterschiede zwischen Plattformen 1 und 2 zu messen, sowie Unterschiede, die hauptsächlich durch eine andere Prozessorarchitektur bedingt sind (zwischen 2 und 3). Als Testdatei wurde in allen Experimenten die zur Verfügung gestellte Audiodatei "nicht_zu_laut_abspielen.wavünd eine Blocksize

[1] `https://numpy.org/doc/stable/reference/generated/numpy.fft.fft.html`
[2] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.io.wavfile.read.html`

von 2205 verwendet. Alle Experimente wurden 3 Mal durchgeführt. Zur besseren Vergleichbarkeit der Ergebnisse wird in allen Experimenten die angesprochene Numpy Implementierung der Fourieranalyse verwendet, die für alle Systeme verfügbar ist. Somit werden algorithmische
Zur Messung des Speicherverbrauchs von Python Skripten stehen u. a. *PSUtil* [3] und *tracemalloc* [4] zur Verfügung. Die "Resident Set Size"(rss), die mithilfe von *PSUtil* erfasst wird, misst den gesamten Speicher, die der entsprechende Prozess im Arbeitsspeicher beansprucht (inkl. Python Interpreter, Heap, Stack und Speicher, der durch Bibliotheken verwendet wird). Im Gegensatz dazu erfasst *tracemalloc* lediglich den Speicher, den das Skript alleine während seiner Laufzeit alloziert. Um herauszufinden, welches Tool eine akkuratere Messung des Speicherverbrauchs ermöglicht, habe ich einen Test in einer vorläufigen Version meines Python Skripts auf System 1 durchgeführt, in dem ich mit beiden Tools den aktuellen Speicherverbrauch alle 3 Sekunden ausgelesen und protokolliert habe. Vergleicht man die Messungen der beiden Tools



(a) Verwendeter Speicher der Fourier-Analyse nach PSUtil

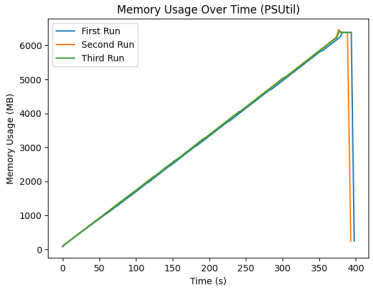


(b) Verwendeter Speicher der Fourier-Analyse nach tracemalloc

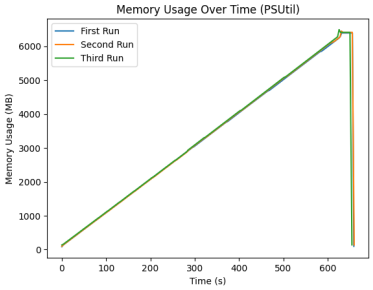
Abbildung 2: Vergleich des verwendeten Speichers einer früheren Version der Fourier-Analyse nach PSUtil und tracemalloc auf System 1. Es wird jeder Block analysiert und alle Statistiken werden in einer Liste aufbewahrt.

(Abbildung 2), so fällt auf, dass PSUtil teilweise treppenartiges Wachstum misst, während tracemalloc ausschließlich lineares Wachstum meldet. Dabei ist das **treppenartige Wachstum** durchaus zu erwarten, da wichtige Python Strukturen wie Listen oder Dictionaries um einen **multiplikativen Faktor** wachsen, wenn sie voll sind, was in einem sprungartigen Anstieg des Speichers zu sehen sein sollte. Die absolute Größe dieser Sprünge wird zudem nach PSUtil wie zu erwarten immer größer. Des Weiteren fällt die deutlich **höhere Höchstauslastung** von ca. 9GB nach PSUtil gegenüber ca. 3,5GB nach tracemalloc auf, wobei sich die Auslastung von PSUtil mit dem Windows Task Manager deckt. Aufgrund dieser Beobachtungen halte ich PSUtil für das akkuratere Tool und werde mich im Folgenden ausschließlich auf diese Messungen beziehen. Zusätzlich ermöglicht dies eine deutlich schnellere Ausführung, da die Laufzeit durch Nutzung von tracemalloc ca. 2x länger (Linux) bzw. 5x länger (Windows) dauert.
Bemerkung. Wie in Abbildung 2 zu sehen ist, kann das Python-Skript viel Speicher benötigen, um die gesamten Fourier Statistiken aller Blöcke in einer Liste aufzubewahren, die erst nach dem gesamten Durchlauf in eine Datei geschrieben wird. Alternativ bietet sich auch eine Implementierung an, in der nach jedem Block die Fourier Statistiken in eine Datei geschrieben werden, sodass die oben genannte Liste entfällt. Diese Implementierung benötigt auf allen Systemen etwa 250MB Speicher, führt jedoch zu einem extremen IO-Bottleneck (Durchsatz reduziert sich auf System 1 um 80%). Als Kompromiss bietet meine Implementierung die Möglichkeit, die Fourier Analyse der Audiodatei in k Chunks mit jeweils $\frac{1}{k}$ der Blöcke durchzuführen und die Fourier Statistiken nach der Verarbeitung jedes Chunks zu speichern, um den Speicherverbrauch zu reduzieren.
Auf allen drei Systemen (Abbildung 3) ist das Verhalten des Speicherverbrauchs sehr ähnlich und über alle drei Durchgänge sehr konstant. Er wächst die meiste Zeit linear, da die Liste, welche die Statistiken der Fourier-Analyse enthält, Block für Block befüllt wird. Der kleine Speicherpeak am Schluss ist auf das Schreiben der CSV-Datei mit den Fourier Statistiken zurückzuführen.

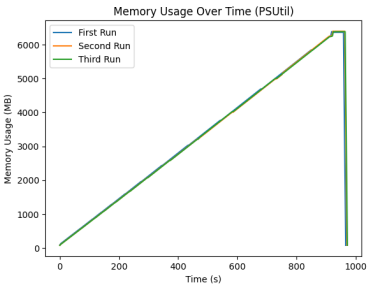
[3] `https://psutil.readthedocs.io/en/latest/index.html`
[4] `https://docs.python.org/3/library/tracemalloc.html`



(a) Speicherverbrauch der Fourier-Analyse auf System 1



(b) Speicherverbrauch der Fourier-Analyse auf System 2



(c) Speicherverbrauch der Fourier-Analyse auf System 3

Abbildung 3: Vergleich des verwendeten Speichers der Fourier-Analyse nach PSUtil auf Systemen 1, 2, 3

Beim Vergleich der genauen Start- und Höchstverbrauchszahlen (Tabelle 1) fällt auf, dass System 3 (Raspberry Pi) in allen Szenarien einen geringeren Speicherbedarf im Vergleich zu beiden anderen Systemen hat. Dies könnte daran liegen, dass Python auf PiOS / einem ARM System eine leicht ressourcenschonendere Standardkonfiguration verwendet und z. B. Listen um einen kleineren multiplikativen Faktor wachsen lässt o. ä. System 2 weist einen etwas höheren Speicherverbrauch (Start- und Endverbrauch) auf als System 1, was durch eine großzügigere Speicherallozierung begründet sein könnte, da dieses System mit Abstand um die größten Ressourcen verfügt.

System	System 1	System 2	System 3
Startverbrauch (MB)	86	108	**81**
Höchstverbrauch (MB)	6422	6445	**6392**
Größe der Executable (MB)	129	**104**	105

Tabelle 1: Vergleich weiterer Kennzahlen der drei Systeme. Ergebnisse beschreiben den Mittelwert aus 3 Durchläufen.

Zusätzlich zum (Arbeits-)Speicherverbrauch der Systeme sind in Tabelle 1 die Größen von standalone Executables aufgeführt, die mit Hilfe von PyInstaller[5] auf den jeweiligen Systemen erstellt wurden und sowohl den Python Interpreter als auch alle benötigten Bibliotheken enthalten. Die Executable ist spezifisch für das entsprechende Betriebssystem, die dort verwendete Wordsize und die verwendete Python Version. Zwischen den Größen für System 3 und System 2 liegt kein nennenswerter Unterschied, was daran liegen sollte, dass sowohl Ubuntu (System 2) als auch PiOS (System 3) 64-bit Debian-Derivate sind. Die Executable für System 1 (Windows) ist merklich größer. Dies könnte daran liegen, dass die verwendeten Bibliotheken für Windows Systeme aufgrund zusätzlicher Windows-spezifischer Optimierungen größer sind.

[5] `https://pyinstaller.org/en/stable/`