Übung 2

Repository: https://github.com/Raphael90K/HCA2

1 Aufgabe 1

Um die erste Aufgabe zu lösen werden zwei Ansätze verfolgt. Zum einen wird eine Basisimplementierung in Python erstellt, die die diskrete Fourier Transformation (FFT) von `numpy`[1] verwendet. Dies soll eine Vergleichbarkeit der Implementierung zwischen den verschiedenen Aufgaben gewährleisten. Zum anderen soll eine Fourier Transformation in Mojo[2] implementiert werden, die mit der Pythonversion verglichen werden soll. Bei Mojo handelt es sich um eine relativ neue Programmiersprache, die eine volle Python-Integration ermöglichen und dennoch enorme Performant sein soll. In einem Beispielprogramm wird eine Beschleunigung um das 68-tausendfache im Vergleich zu Python und sogar eine Leistungssteigerung im Vergleich zu C++ erreicht.[Modu24] Da es bisher kaum Bibliotheken für Mojo gibt, musste die Fourier Transformation neu implementiert werden. Die die komplexe Division und Exponentialfunktion werden aktuell ebenfalls noch nicht unterstützt und mussten daher ebenfalls implementiert werden. Um eine Vergleichbarkeit zu gewährleisten, wurde der rekursive Algorithmus von Cooley und Tukey zur Berechnung der FFT verwendet [CoTu65]. Dieser wird auch von `numpy` genutzt[3]. Bei der Implementierung der Berechnungen wurde ChatGPT[4] unterstützend verwendet, wobei lediglich Pseudocode als Grundlage generiert werden konnte, da ChatGPT in der genutzten Version mit Mojo kaum zurecht kam.
Für die Ausgabe wurde eine `.wav` Datei mit der Länge eine Minute und den Frequenzen 42, 420 und 4200 Hz verwendet. Die Blockgröße beträgt 256 Samples und der Offset 1 Sample. Die nachfolgenden Ausgaben wurden mit Jupyter Notebooks[5] erstellt.

[1] https://numpy.org/doc/stable/reference/generated/numpy.fft.fft.html#numpy.fft.fft
[2] https://www.modular.com/mojo
[3] https://numpy.org/doc/stable/reference/generated/numpy.fft.fft.html#numpy.fft.fft
[4] https://chatgpt.com/
[5] https://jupyter.org/

[2]:
```
%%python
from A1 import A1Python

sample_rate, audio_data = A1Python.read_wave_file('./Audios/output.wav')
durationA1Python = A1Python.calculate(audio_data, sample_rate, block_size=256,␣
↪offset=1, threshold=1_000_000)
print(durationA1Python)
```

```
Frequency: 0.00 Hz, Amplitude: 1641160.11
Frequency: 344.53 Hz, Amplitude: 1029072.78
Frequency: 4134.38 Hz, Amplitude: 1105567.05
8.284553050994873
```

[3]:
```
import A1.A1Mojo as A1Mojo
from python import Python, PythonObject
from complex import ComplexFloat64

var readwav = Python.import_module("utils.utils")

var input = readwav.read_wave_file('./Audios/output.wav')
var sample_rate = input[0].to_float64()
var data_input = input[1]
var audio = A1Mojo.ndarray_to_complexlist(data_input)

var ampA1 = A1Mojo.amplitude(256 // 2, audio, sample_rate, 256, 1)
var durationA1Mojo = A1Mojo.calculate(ampA1, 256, 1000000)
print(durationA1Mojo)
```

```
Freq:  0.0  :  1641160.1144910036
Freq:  344.53125  :  1029072.7470971501
```

```
Freq:  4134.375  :  1105567.0263031418
255.89522617
```

Python erreichte hierbei eine Laufzeit von 8.2846 Sekunden und Mojo von 255.8952 Sekunden. Die serielle Version von Mojo brauchte für die Ausführung also fast 31 mal so Lange wie die Python. Dies dürfte daran liegen, dass bei der Implementierung der Mojo Version der FFT Listenstrukturen verwendet wurden und `numpy` zu großen Teilen in performantem C geschrieben ist[6]. Die Beschleunigung des Mojo Codes war jedoch in der kurzen Zeit nicht möglich.

2 Aufgabe 2

Die Implementierung zur Erzeugung von `.wav` Dateien wurde mit Hilfe des `wave`[7] Module erstellt. Der Nutzer kann über die Kommandozeile die Länge sowie beliebige Frequenz / Amplituden Paare wählen. Die Frequenz wird als Parameter in einer Sinusfunktion verwendet und mit der gewählten Amplitude multipliziert. Die gewählten Frequenzen werden dann nacheinander aufsummiert und somit überlagert.

3 Aufgabe 3

In dieser Aufgabe wurde die Python und Mojo Lösung derart verändert, dass eine parallele Ausführung des Codes möglich ist. Die verwendeten Funktionen zur Berechnung der FFT und der grundsätzliche Programmablauf verhält sich gleich zu der Implementierung in Aufgabe 1. Für die Python Implementierung wurde das Modul `multiprocessing`[8] verwendet. Die Idee der Implementierung war, dass eine Anzahl an Prozessen erstellt werden die die FFT lokal berechnen und die Ergebnisse Ihrer zugewiesenen Blöcke lokal aufsummieren. Sobald alle Berechnungen abgeschlossen sind, werden die lokalen Ergebnisse aufsummiert und durch die Gesamtzahl der Blöcke dividiert. So soll ein ausgewogenes Verhältnis zwischen Speicherplatznutzung und Transfers zwischen Pro-

[6] https://github.com/numpy/numpy
[7] https://docs.python.org/3/library/wave.html
[8] https://docs.python.org/3/library/multiprocessing.html

zessen entstehen. Ein Datarace soll so ebenfalls verhindert werden.

Die Mojo Implementierung nutzt die Funktion `parallelize`[9], in der eine Funktion mit einem Integer Argument als Parameter verwendet wird. Die Parameter werden vom Mojo Compiler zur Compilezeit ausgewertet und sind daher statisch[10]. Bei den Experimenten fiel zudem auf, dass bei der Ausführung höchstwahrscheinlich Raceconditions auftreten, da die Ergebnisse bei gleichen Eingaben unterschiedlich waren. Die Möglichkeit der Verwendung eines Locks konnte in Mojo bisher nicht gefunden werden und bei dem Versuch Python Locks zu verwenden, kam es zu Fehlern, die nicht behoben werden konnten. Obwohl Mojo ein Rust ähnliches Ownership und Borrowing besitzen sollte, scheint dies auch ausweislich der Dokumentation noch nicht implementiert[11]. Die Versuche wurden mit den gleichen Eingaben wie in Aufgabe 1 durchgeführt und führten zu folgenden Ergebnissen.

[4]:
```
%%python
from A3 import A3Python


durationA3Python = A3Python.calculate(audio_data, sample_rate, block_size=256,␣
 ↪offset=1, threshold=1_000_000)
print(durationA3Python)
```

```
Cores used: 32
Frequency: 0.00 Hz, Amplitude: 1641159.91
Frequency: 344.53 Hz, Amplitude: 1029072.75
Frequency: 4134.38 Hz, Amplitude: 1105567.05
1.312391996383667
```

[5]:
```
import A3.A3Mojo as A3Mojo


var inputA3 = readwav.read_wave_file('./Audios/output.wav')
```

[9] https://docs.modular.com/mojo/stdlib/algorithm/functional/parallelize
[10] https://docs.modular.com/mojo/manual/parameters/
[11] https://docs.modular.com/mojo/manual/values/ownership

```
var sample_rateA3 = input[0].to_float64()
var data_inputA3 = input[1]
var audioA3 = A3Mojo.ndarray_to_complexlist(data_input)

var ampA3 = A3Mojo.ParallelFft(audio, sample_rate, 256, 1)
durationA3Mojo = A3Mojo.calculate(ampA3, 1000000)
print(durationA3Mojo)
```

```
Freq:  0.0  :  1640871.0886665527
Freq:  344.53125  :  1028804.1556153116
Freq:  4134.375  :  1104516.9023475153
48.426688403999997
```

Bei dieser Aufgabe war die Python Implementierung mit 1,3124 Sekunden wieder ca. 37 mal schneller als das Mojo Programm mit 48,4267 Sekunden. Im Vergleich zu Aufgabe 1 konnte mit Python ein Speedup von 6,31 und mit Mojo ein Speedup von 5,28 erreicht werden. Da das eingesetzte System über 32 Threads verfügt, konnte ein Speedup von k nicht erreicht werden.

4 Aufgabe 4

Für die Berechnungen auf der Grafikkarte wurden Funktionen von `cupy`, welches das CUDA Toolkit verwendet, genutzt[12]. `Cupy` verwendet zudem ähnliche Funktionen zu `numpy`, was die Vergleichbarkeit erhöht[13]. Folgendes Vorgehen soll die Laufzeit optimieren:

1. Die komplette Audiodatei wird in den Grafikspeicher kopiert, dies soll die Transfers zwischen CPU und GPU minimieren.
2. Die Verarbeitung erfolgt in Blöcken (Batches).
3. Das Auslesen der maximalen Threads der verwendeten Grafikkarte ergab 55296 Threads.

[12] https://cupy.dev/
[13] https://docs.cupy.dev/en/stable/reference/fft.html

Dies wird als Batchsize verwendet und lieferte bei unstrukturierten Tests auch die besten Ergebnisse.

4. Um die einzelnen Blöcke zur Berechnung nicht kopieren zu müssen wurde mit der Funktion `cp.lib.stride_tricks.as_strided`[14] eine View auf die Audiodatei erzeugt.

5. Das Aufsummieren der Ergebnisse erfolgt ebenfalls auf der Grafikkarte, erst nach Berechnung des Durchschnitts erfolgt ein Transfer zurück in den Hauptspeicher.

Bei den gleichen Eingaben wie in Aufgabe 1 wurden folgendes Ergebnis erzielt.

[6]:
```
%%python
from A4 import A4Batch


durationA4Python = A4Batch.calculate(audio_data, sample_rate, block_size=256,␣
 ↪offset=1, threshold=1_000_000, batch_size=55296)
print(durationA4Python)
```

```
Batchgröße:  55296
Frequency: 0.00 Hz, Amplitude: 1641160.11
Frequency: 344.53 Hz, Amplitude: 1029072.78
Frequency: 4134.38 Hz, Amplitude: 1105567.05
1.1822559833526611
```

Die Laufzeit von 1,1823 stellt lediglich eine Beschleunigung von 1,11 dar. Abbildung 1 stellt die verschiedenen Laufzeiten der Aufgaben noch einmal dar.

[14] https://docs.cupy.dev/en/stable/reference/generated/cupy.lib.stride_tricks.as_strided.html

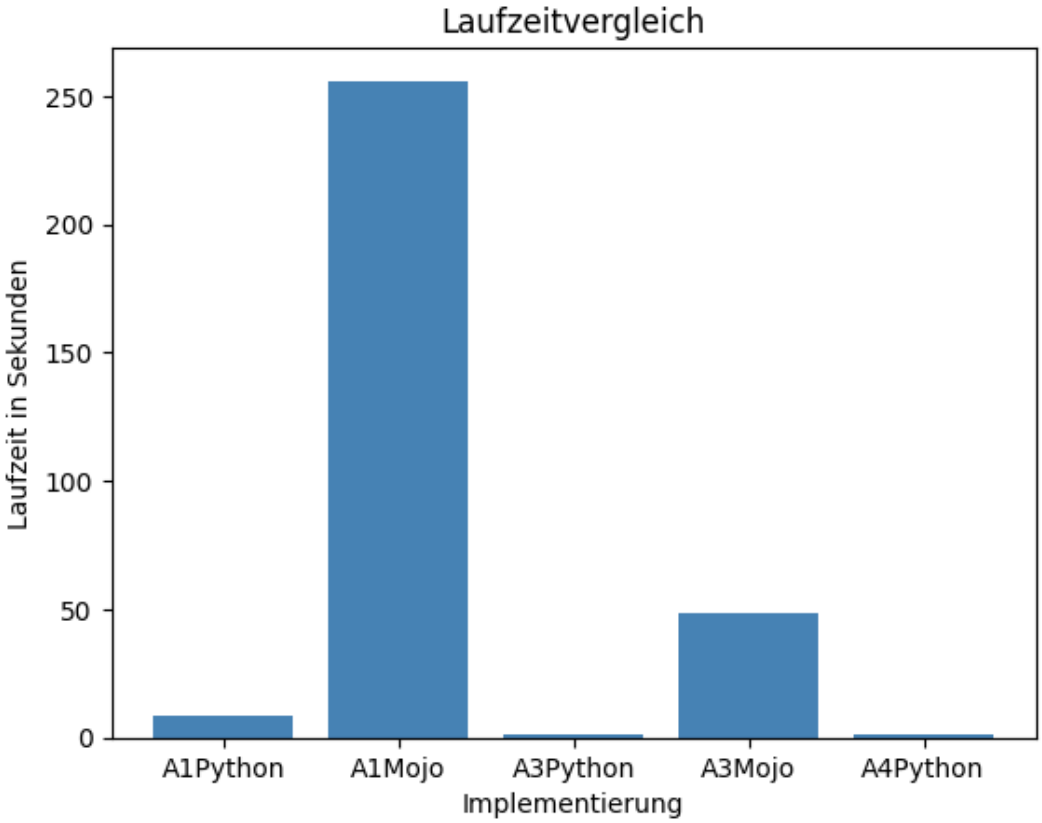

Abbildung 1: Laufzeitvergleich der verschiedenen Implementierungen

5 Vergleich der Python Implementierungen

Zum Vergleich der Laufzeiten wurde die Serielle-, Parallele- und GPU-Implementierung von Python auf einer 10 minütigen Audiodatei mit den Frequenzen 42, 420 und 4200 getestet und verglichen. Dies soll zeigen, ob es bei größeren Dateien zu deutlicheren Unterschieden kommt. Hierbei wurde eine Blockgröße von 1024 verwendet.

[8]:
```
%%python
print('seriell')
sample_rate, audio_data = A1Python.read_wave_file('./Audios/sound10minutes.
  ↪wav')
```

```
durationA1Python10m = A1Python.calculate(audio_data, sample_rate,␣
  ↪block_size=1024, offset=1, threshold=1_000_000)
print(durationA1Python10m, '\n')

print('parallel')
durationA3Python10m = A3Python.calculate(audio_data, sample_rate,␣
  ↪block_size=1024, offset=1, threshold=1_000_000)
print(durationA3Python10m, '\n')

print('grafikbeschleunigt')
durationA4Python10m = A4Batch.calculate(audio_data, sample_rate,␣
  ↪block_size=1024, offset=1, threshold=1_000_000, batch_size=55296)
print(durationA4Python10m, '\n')
```

```
seriell
Frequency: 43.07 Hz, Amplitude: 5680271.66
Frequency: 387.60 Hz, Amplitude: 1688562.34
Frequency: 430.66 Hz, Amplitude: 5128267.45
Frequency: 473.73 Hz, Amplitude: 1018961.71
Frequency: 4134.38 Hz, Amplitude: 1184162.51
Frequency: 4177.44 Hz, Amplitude: 3444567.31
Frequency: 4220.51 Hz, Amplitude: 3789016.70
Frequency: 4263.57 Hz, Amplitude: 1222348.95
205.89144015312195

parallel
Cores used: 32
Frequency: 43.07 Hz, Amplitude: 5680271.66
Frequency: 387.60 Hz, Amplitude: 1688562.34
```

```
Frequency: 430.66 Hz, Amplitude: 5128267.45
Frequency: 473.73 Hz, Amplitude: 1018961.71
Frequency: 4134.38 Hz, Amplitude: 1184162.51
Frequency: 4177.44 Hz, Amplitude: 3444567.31
Frequency: 4220.51 Hz, Amplitude: 3789016.70
Frequency: 4263.57 Hz, Amplitude: 1222348.95
18.853911876678467

grafikbeschleunigt
Batchgröße:  55296
Frequency: 43.07 Hz, Amplitude: 5680271.66
Frequency: 387.60 Hz, Amplitude: 1688562.34
Frequency: 430.66 Hz, Amplitude: 5128267.45
Frequency: 473.73 Hz, Amplitude: 1018961.71
Frequency: 4134.38 Hz, Amplitude: 1184162.51
Frequency: 4177.44 Hz, Amplitude: 3444567.31
Frequency: 4220.51 Hz, Amplitude: 3789016.70
Frequency: 4263.57 Hz, Amplitude: 1222348.95
12.569470643997192
```

Hier ist zu erkennen, dass der Speedup bei der größeren Datei mit größerer Blockgröße zunimmt. Abbildung 2 stellt die Laufzeiten grafisch dar.

6 Fazit

In den verschiedenen Aufgaben konnte gezeigt werden, dass mit der parallelen Lösung und der GPU Implementierung ein deutlicher Speedup bei der Berechnung der FFT erzielt werden konnte. Dieser Speedup lag jedoch deutlich unter den vom System möglichen Threads. Die Implementierung in

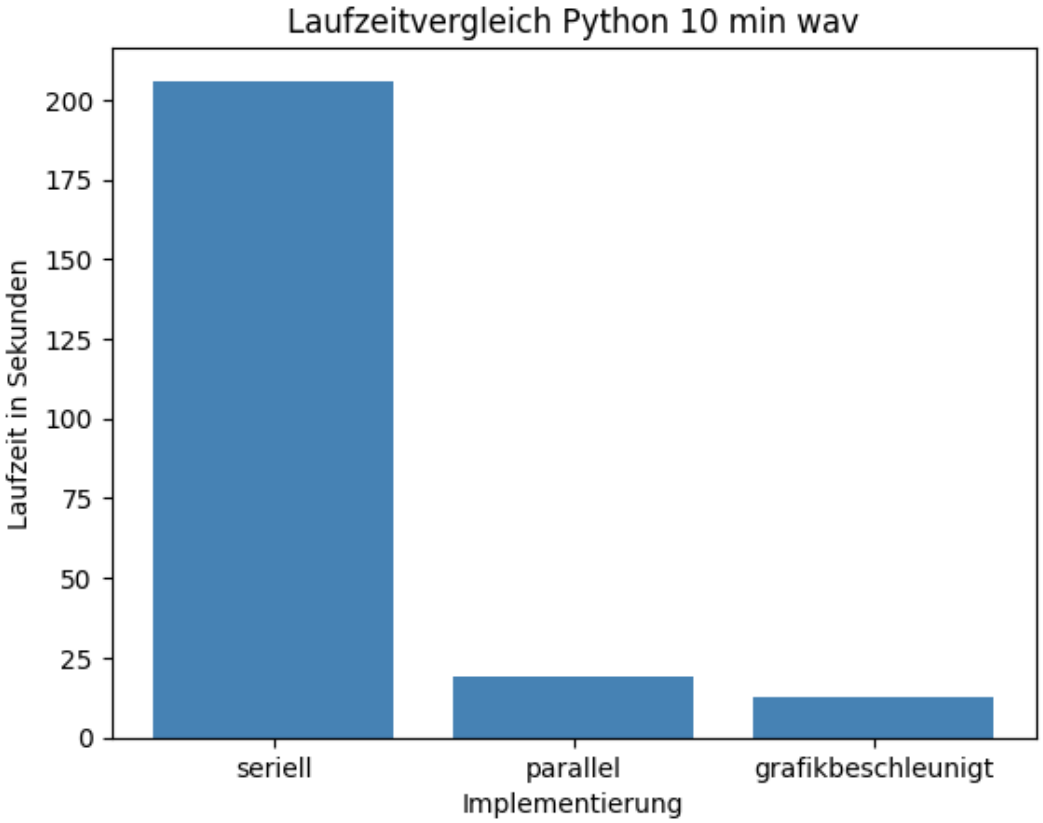

Abbildung 2: Laufzeitvergleich 10 Min. `.wav` mit Blockgröße 1024

Mojo konnte mit dem aktuellen Stand nicht an die Performanz der Pythonlösung heranreichen. Dies lag höchstwahrscheinlich an der ineffizienten Implementierung. Ein Upgrade war jedoch in der kurzen Zeit nicht möglich, dies hatte folgende Gründe. Da es sich um eine relativ junge Programmiersprache handelt, gibt es kaum effiziente Bibliotheken die verwendet werden können. Diese müssen selbst implementiert werden. Zudem gibt es zwar ein Plugin für VS-Code[15], dieses verfügt jedoch nur über eingeschränkte Codevervollständigung. Da die Implementierung durch die zuvor genannten Parameter statische Komponenten aufweist, kommt es häufiger zu Kompilierungs- oder Laufzeitfehlern, die jedoch teils gar nicht aufzeigen, woher der Fehler resultiert. Die Verwendung von Python Bibliotheken ist möglich, jedoch kam es bei der Bearbeitung der Aufgaben häufig zu unerklärlichen Abstürze der Laufzeitumgebung, die auch nicht behoben werden konnten und die Ursache nicht darstellten. Daher gestaltete sich die Entwicklung schwierig.

[15] https://marketplace.visualstudio.com/items?itemName=modular-mojotools.vscode-mojo

Literatur

[CoTu65] J. W. Cooley und J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation* 19(90), 1965, S. 297–301.

[Modu24] Modular. Mojo: Programming language for all of AI. https://www.modular.com/mojo, 2024. Stand: 13.07.2024, Aufruf am: 13.07.2024.