Jan Niclas Ruppenthal
1481198

Heterogenous Computing – Übung 02

Hilfsfunktionen für alle Aufgaben

```
def main(analyze_method):
    file_path, block_size, shift_size, threshold = get_all_arguments()
    wav_data, sample_rate = analyze_wav_file(file_path)

    start_time = time.time()

    aggregated_fft = analyze_method(wav_data, block_size, shift_size)

    run_time = time.time() - start_time
    print_run_time(run_time)

    write_data_to_files(aggregated_fft, wav_data, sample_rate, block_size, threshold)
    print_results(aggregated_fft, sample_rate, block_size, threshold)
```

Die Datei **util.py** im Ordner **FFT** definiert alle Funktionen, die in den Aufgaben 1, 3 sowie 4 benutzt werden. In der obigen Hauptmethode wird zuerst alle übergebenen Parameter beim Starten des Programms gelesen. Danach wird die WAV-Datei gelesen und es wird der Zeitpunkt nach dem Lesen der Audiodatei gespeichert. Mithilfe der gelesenen Parameter und Daten der Audiodatei wird die FFT-Analyse gestartet. Welche Methode hierbei aufgerufen wird, ist abhängig vom Parameter der **main**-Methode. In jeder Implementierung der Aufgaben 1, 3 und 4 wird diese Hauptmethode aufgerufen, wobei die Funktion zur Fourieranalyse übergeben wird. Durch dieses Vorgehen konnte ich duplizierten Code vermeiden. Im Anschluss wird die Zeit gestoppt, da wir an der Zeit für eine Fourieranalyse interessiert sind. Nach der Fouriertransformation werden erneut die wichtigsten Daten und Ergebnisse in drei verschiedene Dateien geschrieben. Mithilfe der Daten in den Dateien können wieder Diagramme mithilfe von **Plot_Data/plotting.py** erstellt werden.

```
def __get_all_arguments():
    file_path = str(sys.argv[1])
    block_size = int(sys.argv[2])

    if block_size < 64:
        block_size = 64
    elif block_size > 512:
        block_size = 512

    shift_size = int(sys.argv[3])
    if shift_size < 1:
        shift_size = 1
    elif shift_size > block_size:
        shift_size = block_size

    threshold = float(sys.argv[4])

    if threshold < 0:
        threshold = 0

    return file_path, block_size, shift_size, threshold
```

Aus der Aufgabenstellung geht hervor, dass wir vier Parameter lesen und beschränken sollen. Dies wird in der Methode **get_all_arguments** ausgeführt. Dabei wird die Blockgröße (**block_size**) auf [64, 512] und der Versatz (**shift_size**) auf [1, **block_size**] eingeschränkt. Zusätzlich habe ich auch noch den Schwellwert auf natürliche Fließkommazahlen begrenzt. Zum Schluss werden alle Parameter zurückgegeben.

```python
def __analyze_wav_file(file_path):
    sample_rate, data = wavfile.read(file_path)

    if len(data.shape) > 1:
        data = data[:, 0]

    return data, sample_rate
```

Beim Lesen der Audiodatei wird noch überprüft, ob das Audiosignal Stereo ist. Die Hälfte der Daten kann ignoriert werden, wenn die Datei aus zwei Kanälen besteht.

```python
def __print_results(aggregated_fft, sample_rate, block_size, threshold):
    result = [(index * sample_rate / block_size, aggregated_fft[index])
              for index in range(len(aggregated_fft)) if aggregated_fft[index] > threshold]

    print(result)
```

Am Ende der **main**-Methode wird noch eine Ausgabe hinzugefügt, die Tupel von Frequenzen und ihre Amplituden in die Konsole schreibt. Dabei werden nur Frequenzen berücksichtigt, deren Amplitudenmittelwert größer ist als der übergebene Schwellwert.

Implementierung der Aufgabe 01

Die Aufgabe wurde in der Datei **FFT/fft_seq.py** implementiert. Für die Implementierung der ersten Aufgabe der zweiten Übung habe ich meine Python-Implementierung aus der letzten Übung angepasst. Hierbei musste ich nur das Lesen der übergebenen Parameter beim Start des Programms ändern und die FFT-Analyse, da wir nun mit einem Versatz zwischen den Datenblöcken in FFT arbeiten sollen. Des Weiteren habe ich auch die anderen FFT- oder DFT-Implementierungen aus dem Quellcode gelöscht.

```python
def analyze(data, block_size, shift_size):
    num_samples = len(data)
    num_blocks = (num_samples - block_size) // shift_size + 1

    aggregated_fft = np.zeros(block_size//2)
    for i in range(0, len(data) - block_size + 1, shift_size):
        block = data[i:i+block_size]
        fft_result = np.fft.fft(block)

        aggregated_fft += np.abs(fft_result[:block_size//2])

    aggregated_fft /= num_blocks

    return aggregated_fft
```

In der Methode **analyze** wird die FFT auf den Daten der WAV-Datei durchgeführt. Diese Daten werden Block für Block über die FFT-Implementierung von *numpy* analysiert. Hierbei werden die Ergebnisse in **aggregated_fft** summiert und nach der for-Schleife wird für jede Frequenz den Mittelwert ihrer Amplituden berechnet.

Implementierung der Aufgabe 02

Der Generator einer WAV-Datei wurde in **Generate/WAV_Generator.py** implementiert. Dabei besteht die Datei aus drei Hauptbestandteilen: die Hauptmethode, Funktionen für 11 verschiedene Testszenarien und das Speichern der generierten Daten in eine Audiodatei.

```python
def main():
    global sample_rate, duration
    filename, func_name, duration, sample_rate, frequencies = get_all_arguments()

    global timestamps
    timestamps = np.linspace(0, duration, int(sample_rate * duration), endpoint=False)

    data = generate_data(func_name, frequencies)
    save_wave_file(filename, data)
```

In der **main**-Methode werden zuerst die Argumente Dateiname, Funktionsname, Dauer, Abtastrate und Frequenzen (oder sonstige notwendige Werte) und in die entsprechenden Variablen gespeichert. Danach wird ein *numpy*-Array von gleichmäßig verteilten Zahlen von 0 bis **duration** erstellt. Dabei hat dieses Array insgesamt eine Länge von **sample_rate * duration**. Die Variable **timestamps** wird von den Funktionen zur Generierung der Daten verwendet, um erneut ein Array von Daten der gleichen Länge zu erstellen. Zum Schluss werden die Daten in eine WAV-Datei geschrieben.

Insgesamt können elf verschieden Testszenarien erstellt werden:

1. Sinuskurve
2. Additive Sinuskurven
3. Amplitudenmodulation
4. Frequenzmodulation
5. Segmenten von Sinuskurven
6. Sinuskurve mit einer Hüllkurve
7. Sinuskurve mit beliebig vielen Harmonischen
8. Dreieckssignal
9. Viereckssignal
10. Zirpe (Signal mit Start- bis Endfrequenz)
11. Noise (zufälliges Signal)

In der Datei HC_Audiodateien.pdf werden alle Funktionen und die Signale dargestellt. In diesem Ergebnisbericht werden nur zwei Implementierungen betrachtet:

```python
def sine_wave(frequency):
    return 0.5 * np.sin(2 * np.pi * frequency * timestamps)


def add_sine_wave(frequencies):
    signal = np.zeros_like(timestamps)
    for frequency in frequencies:
        signal += 0.5 * np.sin(2 * np.pi * frequency * timestamps)
    return signal
```

In der Methode **sine_wave** werden Daten eines Sinussignals einer bestimmten Frequenz erstellt. Die nächste Methode generiert verschiedene Sinuskurven mit diversen Frequenzen, die miteinander addiert werden.

In jeder Methode werden die Daten mit 0.5 multipliziert, damit die Signalwerte zwischen [-0.5, 0.5] liegen. Dadurch wird sichergestellt, dass der Bereich zwischen [-1, 1] nicht überschritten wird. Zudem wird damit auch Clipping vermieden. Clipping tritt auf, wenn der maximale oder minimale Wert über- oder unterschritten wird. Des Weiteren kann durch die Multiplikation von 0.5 die Wahrscheinlichkeit verringert werden, dass das Signal beim Speichern in eine Datei abgeschnitten wird.

```
def save_wave_file(filename, data):
    scaled = np.int16(data / np.max(np.abs(data)) * 32767)
    write(filename, sample_rate, scaled)
```

Im letzten Teil des Skripts werden die Signaldaten in eine Datei geschrieben. Hierbei werden die Daten durch **data / np.max(np.abs(data))** zuerst auf den Amplitudenwert 1 normalisiert. Danach werden diese Werte mit dem höchsten Wert 32767 (= 2^{15} - 1) multipliziert, damit die Amplituden im 16Bit-Zahlenbereich liegen.

Implementierung der Aufgabe 03

Die Implementierung dieser Aufgabe befindet sich in der Datei **FFT/fft_par_multiprocess.py**. Für die dritte Aufgabe der zweiten Übung habe ich die Methode **analyze** (siehe oben in Aufgabe 1) folgendermaßen angepasst:

```
def analyze(data, block_size, shift_size):
    num_samples = len(data)
    num_blocks = (num_samples - block_size) // shift_size + 1

    cpu_count = os.cpu_count()
    print("#Prozessoren: %s" % cpu_count)
    processes = []
    lock = mp.Lock()

    aggregated_fft = mp.Array('d', block_size//2)

    for id in range(cpu_count):
        process = mp.Process(target=fft_process,
                             args=(id, lock, cpu_count, data, block_size, shift_size, aggregated_fft))
        processes.append(process)
        process.start()

    for process in processes:
        process.join()

    aggregated_fft = np.frombuffer(aggregated_fft.get_obj())
    aggregated_fft /= num_blocks

    return aggregated_fft
```

Die Initialisierung der ersten drei Variablen bleibt im Vergleich zur sequenziellen Methode gleich. Danach werden die ersten Variablen für die Parallelisierung erstellt. Der Lock wird später verwendet, damit keine Race Conditions entstehen, wenn die Zwischenergebnisse der FFT-Analyse in **aggregated_fft** addiert werden. Mithilfe der for-Schleife werden die Threads mit Zielfunktion sowie Argumenten erstellt und in einer Liste **processes** hinzugefügt. Am Ende dieser Schleife wird der erzeugte Thread gestartet.

Nachdem alle Threads gestartet wurden, wartet der Main-Thread, bis alle anderen Threads ihre FFT auf den Daten der Audiodatei durchgeführt haben. Danach wird der Mittelwert der berechneten Ergebnisse ermittelt. Zum Schluss wird das Ergebnis in **aggregated_fft** zurückgegeben.

```
def fft_process(id, lock, cpu_count, data, block_size, shift_size, aggregated_fft):
    local_fft = np.zeros(block_size//2)
    for i in range(id * shift_size, len(data) - block_size + 1, cpu_count * shift_size):
        block = data[i:i+block_size]
        fft_result = np.fft.fft(block)
        local_fft += np.abs(fft_result[:block_size//2])

    with lock:
        for i in range(block_size//2):
            aggregated_fft[i] += local_fft[i]
```

Die Methode **fft_process** ist die Funktion, die alle Threads ausführen. Da jeder Thread eine eigene ID hat, kann das Striping Verfahren aus der Vorlesung verwendet werden. Jeder Thread führt die Fouriertransformation mit dem Datenblock, der bei **id*shift_size** beginnt. Die Schrittweite ist dabei **cpu_count*shift_size**.

Für eine bessere Visualisierung der Aufteilung der Datenblöcke betrachten wir nun ein Beispiel mit vier Threads, einer Blockgröße von 5, ein Versatz von 1 und die Datengröße von 20:

ID	1. Iteration	2. Iteration	3. Iteration	4. Iteration
0	[0, 4]	[4, 8]	[8, 12]	[12, 16]
1	[1, 5]	[5, 9]	[9, 13]	[13, 17]
2	[2, 6]	[6, 10]	[10, 14]	[14, 18]
3	[3, 7]	[7, 11]	[11, 15]	[15, 19]

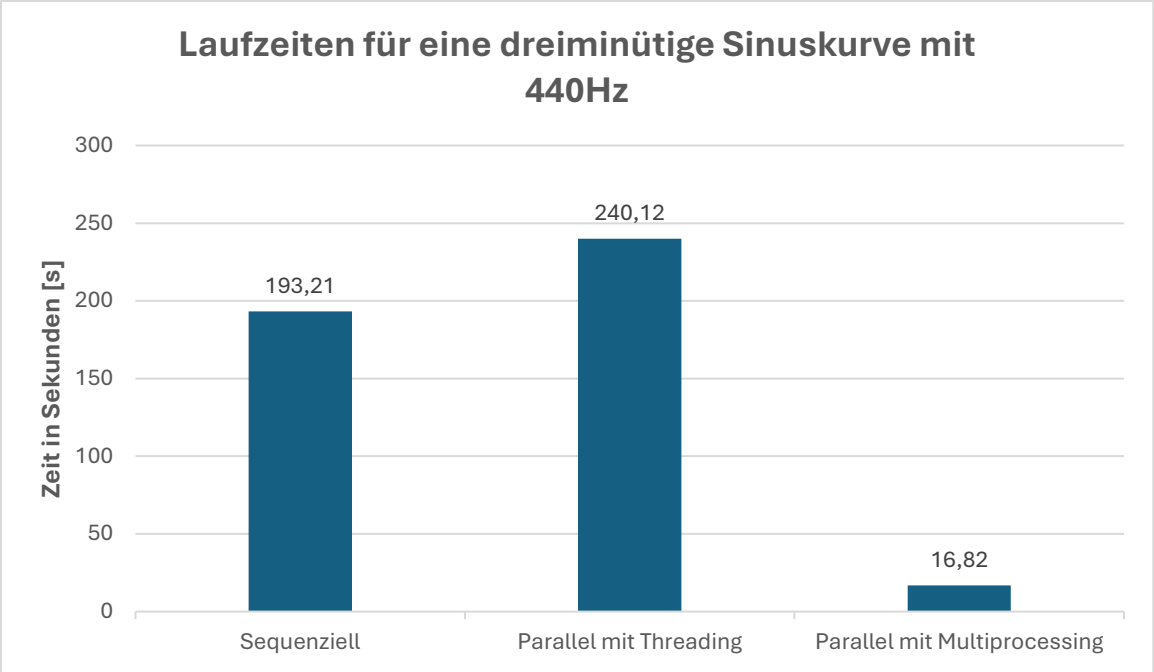
Wenn man nun einen Versatz von drei wählt, dann ergeben sich die folgenden Iterationen:

ID	1. Iteration	2. Iteration
0	[0, 4]	[12, 16]
1	[3, 7]	[15, 19]
2	[6, 10]	
3	[9, 13]	

Am Ende der for-Schleife in **fft_process** werden die Zwischenergebnisse der Fouriertransformation auf den entsprechenden Datenblock **local_fft** addiert. Nach allen Iterationen der Schleife wird das Ergebnis eines Threads mit **aggregated_fft** addiert. Hierbei verwende ich einen Lock, da sonst Race Conditions entstehen können, weil alle Threads auf einer Ressource lesen und schreiben.

Weitere Lösung mit der Bibliothek threading

In meiner ersten Lösung die Fourieranalyse zu parallelisieren, habe ich die Bibliothek *threading* verwendet. Bei meinen ersten Laufzeittests mit einer dreiminütigen Audiodatei (Abtastrate von 44,1kHz), die eine Sinuskurve mit einer Frequenz von 440 Hz enthält, erhielt ich bei einer Blockgröße von 512 und einem Versatz von 1 die folgenden Laufzeiten:



Die parallele Lösung ist schlechter als die sequenzielle Variante. Hierbei habe ich durch eine kleine Recherche herausgefunden, dass die Bibliothek *threading* die Threads nacheinander laufen lässt und es kann nur ein CPU-Prozessor für die Threads verwendet werden[1]. Daher habe ich mich für die Bibliothek *mulitprocessing* entschieden. Die Implementierung ist auch schneller als die sequenzielle Variante.

Aufgrund der langen Laufzeit wird die parallele Implementierung mit *threading* nicht weiter betrachtet.

[1] https://proxiesapi.com/articles/why-is-python-multithreading-slow-and-how-to-speed-it-up (Letzter Zugriff am 12.07.2024)

Jan Niclas Ruppenthal
1481198

Implementierung der Aufgabe 04

Da mein Laptop keine Nvidia Grafikkarte besitzt, habe ich die letzte Aufgabe mit OpenCL implementiert. Die Datei **fft_openCL.py** im Ordner **FFT** implementiert die parallele Lösung, die auf der GPU ausgeführt wird. Hierbei wurde die Methode **analyze** wie in den vorherigen Aufgaben folgendermaßen angepasst:

```
def analyze(data, block_size, shift_size):
    platforms = cl.get_platforms()
    devices = platforms[0].get_devices(cl.device_type.GPU)
    units_count = devices[0].max_compute_units
    print("#Recheneinheiten: %s" % units_count)

    context = cl.create_some_context()
    queue = cl.CommandQueue(context)

    mf = cl.mem_flags
    data = data.astype(np.float32)
    num_blocks = (len(data) - block_size) // shift_size + 1
    max_limit = min(4000000, num_blocks)

    program_source = """
    […]
    """

    program = cl.Program(context, program_source).build()
    fft_kernel = program.fft_kernel

    aggregated_fft = np.zeros(block_size // 2, dtype=np.float32)
    start = 0
    num_blocks_temp = num_blocks
    while num_blocks_temp > 0:
        current_limit = min(max_limit, num_blocks_temp)
        fft_result = np.zeros((current_limit, block_size // 2), dtype=np.float32)
        data_buffer = cl.Buffer(context, mf.READ_ONLY | mf.COPY_HOST_PTR,
hostbuf=data[start:start + current_limit * shift_size + block_size])
        result_buffer = cl.Buffer(context, mf.WRITE_ONLY, fft_result.nbytes)

        fft_kernel.set_args(data_buffer, result_buffer, np.int32(block_size),
                            np.int32(shift_size), np.int32(len(data[start:start +
current_limit * shift_size + block_size])))

        cl.enqueue_nd_range_kernel(queue, fft_kernel, (current_limit,), None)
        cl.enqueue_copy(queue, fft_result, result_buffer).wait()

        aggregated_fft += np.sum(fft_result, axis=0)

        num_blocks_temp -= current_limit
        start += current_limit * shift_size

    aggregated_fft /= num_blocks

    return aggregated_fft
```

Ein besonderer Aspekt dieser Lösung besteht darin, dass die Daten aufgeteilt werden. Diese aufgeteilten Daten werden anschließend auf die GPU übertragen, die dann eine FFT ausführt. Die Datenaufteilung ist entscheidend, da die GPU nur über begrenzten Speicher verfügt. Dabei wird auch verhindert, dass das Programm wegen mangelnder Ressourcen abstürzt. Das momentane Limit 4000000 muss man im Quellcode ändern. Hierbei habe ich diese Zahl gewählt, damit ich im späteren Vergleich der Laufzeit bessere Ergebnisse erhalte. Je kleiner diese Größe **max_limit** ist, desto öfters müssen die Daten auf die GPU kopiert werden.

Am Anfang der Methode wird die Anzahl der Recheneinheiten, die die GPU verfügt, ausgegeben. Danach werden für das Programm Variablen definiert und initialisiert. Hierbei wird die maximale Größe der Daten über **max_limit** festgelegt.

Innerhalb der for-Schleife werden die Daten in jeder Iteration auf die GPU kopiert. Die Ergebnisse der FFT werden von der GPU in ein zweidimensionales Array mit dem Namen **fft_result** geschrieben. Jedes Zwischenergebnis einer Fourieranalyse wird basierend auf der ID der Recheneinheit in die entsprechende Zeile von **fft_result** eingetragen. Des Weiteren muss man wegen der Einteilung der Audiodaten darauf achten, dass keine Daten verloren gehen oder doppelt gelesen werden. Nach der Ausführung auf der GPU wird das Ergebnis von der CPU in **fft_result** summiert und in **aggregated_fft** gespeichert. Dann wird der nächste Startpunkt des nächsten Datenblocks ermittelt. Die Schleife endet, nachdem alle Daten von der GPU bearbeitet wurden.

Der Quellcode, der auf der GPU ausgeführt wird, wird in der Variable **program_source** als Kommentar definiert:

```
__kernel void fft_kernel(__global const float *data, __global float *result, int block_size,
int shift_size, int data_length) {
        int gid = get_global_id(0);
        int block_start = gid * shift_size;
        if (block_start + block_size > data_length) return;

        for (int k = 0; k < block_size/2; k++) {
            float sum_real = 0.0f;
            float sum_img = 0.0f;
            for (int n = 0; n < block_size; n++) {
                float angle = -2.0f * M_PI * k * n / block_size;
                sum_real += data[block_start + n] * cos(angle);
                sum_img -= data[block_start + n] * sin(angle);
            }
            result[gid * block_size/2 + k] = sqrt(sum_real * sum_real + sum_img * sum_img);
        }
    }
```

Hierbei musste ich auf den Algorithmus der DFT verwenden, da jeder Versuch eine FFT in OpenCL zu implementieren, gescheitert ist. Bei diversen FFT-Implementierungen brach das Programm ab und führte auch zum Einfrieren meines Laptops.

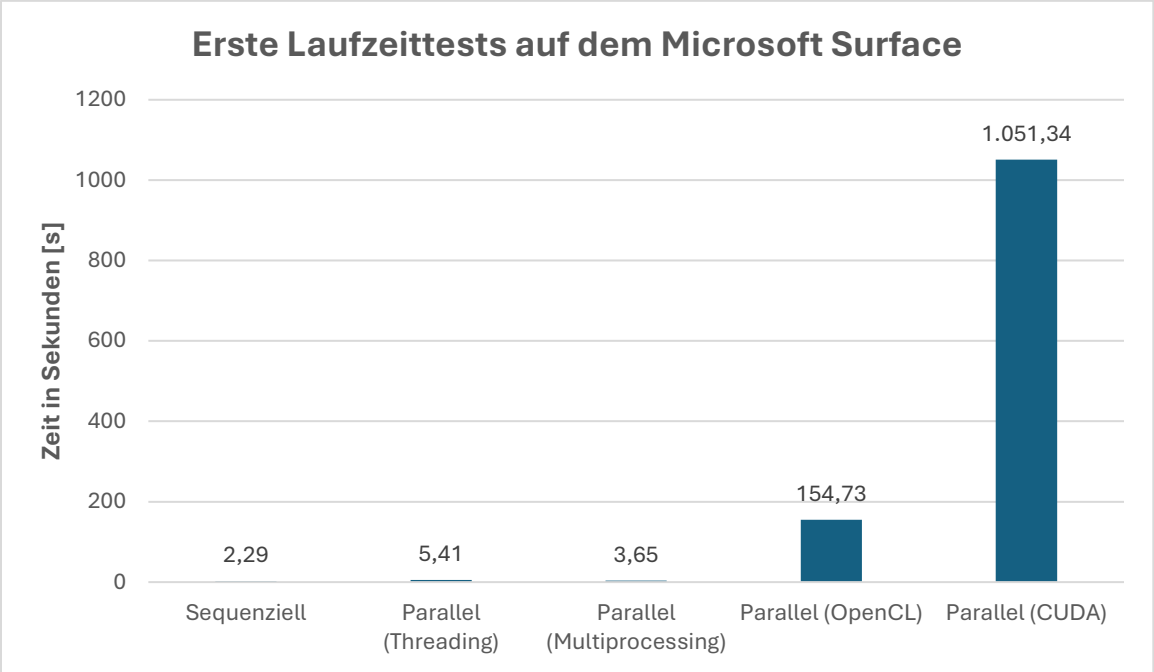
Bei dieser Implementierung wird über eine doppelt verkettete for-Schleife die Formel der diskreten Fouriertransformation auf die Daten angewandt und gespeichert. Mithilfe der Wurzel im letzten Statement wird der Betrag der komplexen Zahl berechnet.

Weitere Lösung mit CUDA

Ich habe vom Lehrstuhl Softwaretechnik einen Microsoft Surface mit einer Nvidia GTX 1050 ausgeliehen bekommen, damit ich eine Fourieranalyse in CUDA implementieren kann. Die Implementierung befindet sich in der Datei **FFT/fft_cuda.py**. Der geliehene Laptop hat die folgenden Spezifikationen:

- Prozessor: Intel(R) Core(TM) i7-8560U @ 1.90 GHz
- Installierter RAM: 16,0 GB
- Integrierte GPU: Intel(R) UHD Graphics 620
- Nvidia GPU: NVIDIA GeForce GTX 1050
- Systemtyp: 64-Bit-Betriebssystem, x64-basierter Prozessor
- Edition: Windows 10 Pro
- Version: 21H2
- Betriebssystembuild: 19044.1288

Bei einer Audiodatei, die eine Sinuskurve mit einer Frequenz von 440 Hz repräsentiert, mit einer Länge von fünf Sekunde, benötigt die Implementierung in CUDA mit einer Blockgröße von 512 und einem Versatz von 1 mehrere Minuten:



Aufgrund der langen Laufzeit wird die Implementierung der Fourieranalyse in CUDA nicht weiter betrachtet.

Vergleich der Laufzeiten der drei Implementierungen

Das Experiment wurde auf meinem Laptop mit den folgenden Spezifikationen durchgeführt:

- Prozessor: Intel(R) Core(TM) Ultra 7 155H 1.40 GHz
- Installierter RAM: 32,0 GB
- Integrierte GPU: Intel(R) Arc(TM) Graphics
- GPU-Speicher: 15 GB
- Systemtyp: 64-Bit-Betriebssystem, x64-basierter Prozessor
- Edition: Windows 11 Home
- Version: 23H2
- Betriebssystembuild: 22631.3880

Insgesamt werden vier verschiedene Experimente durchgeführt. Für diese Experimente wurden die folgenden Dateien erstellt:

1. test01_sine.wav
    a. Funktion auf WAV_Generator.py: sine
    b. Dauer in Sekunden: 720
    c. Abtastrate: 44100
    d. Frequenz: 440

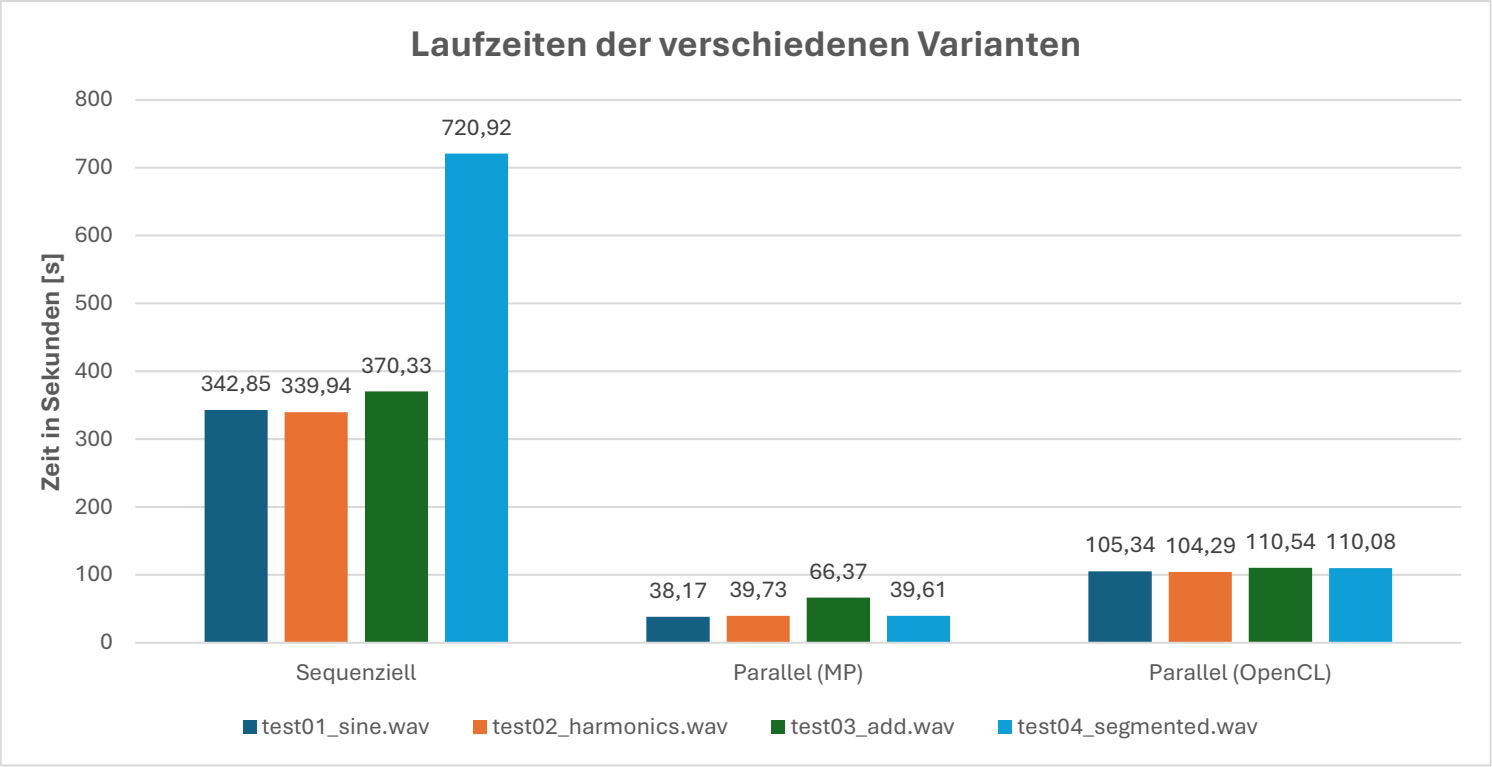
2. test02_harmonics.wav
    a. Funktion in WAV_Generator.py: harmonics
    b. Dauer in Sekunden: 720
    c. Abtastrate: 44100
    d. Frequenz: 500
    e. Anzahl der Harmonischen: 4
3. test03_add.wav
    a. Funktion in WAV_Generator.py: add
    b. Dauer in Sekunden: 720
    c. Abtastrate: 44100
    d. Frequenzen: 300, 600, 900
4. test04_segmented.wav
    a. Funktion in WAV_Generator.py: segmented
    b. Dauer in Sekunden: 720
    c. Frequenzen: 400, 800
    d. Dauer der Segmente: 360

Alle Dateien befinden sich im Ordner **resources**. Dabei wurden für alle Implementierungen (Sequenziell, Parallel mit Mulitprocessing und Parallel mit OpenCL) die folgenden Parameter gewählt:

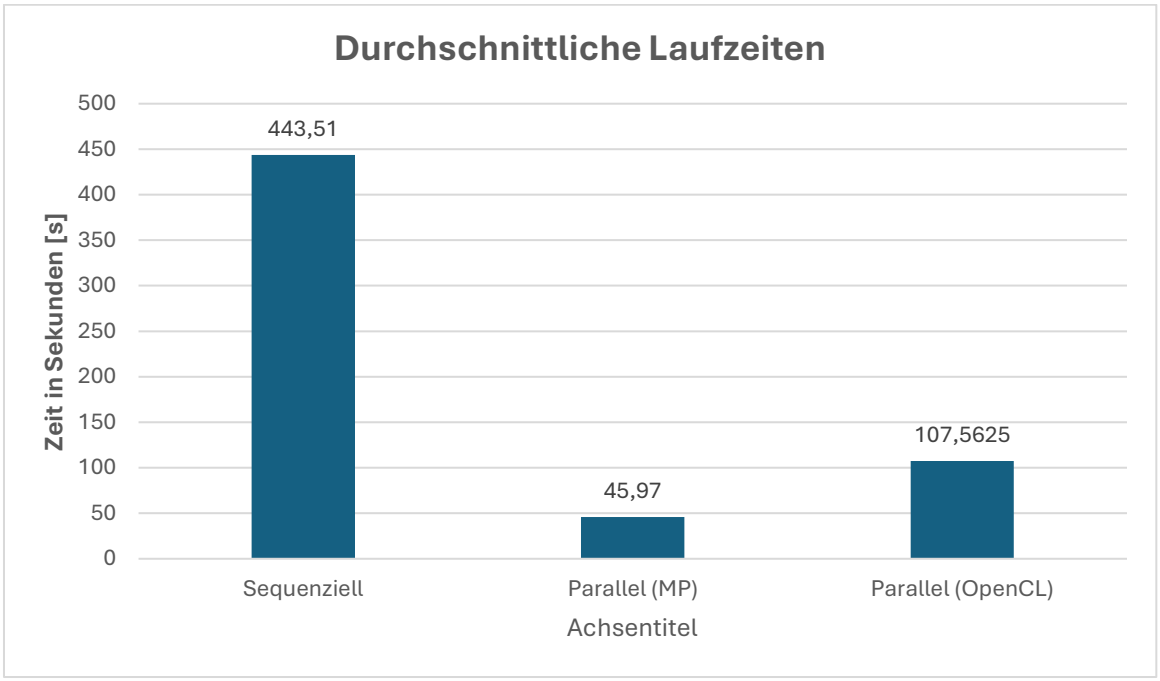
1. Blockgröße: 512
2. Versatz: 1
3. Schwellwert: 200000

Die Laufzeiten der verschiedenen Varianten mit den vier Audiodateien werden in der folgenden Abbildung dargestellt:

Jan Niclas Ruppenthal
1481198



Für einen besseren Vergleich werden nun die durchschnittlichen Laufzeiten betrachtet:



Der Prozessor besitzt 16 Kerne und die parallele Implementierung der FFT hat einen Speedup von 9,65 (= 443,51 / 45,97).

Meine GPU besitzt 8 X^e Kerne und die parallele Implementierung der Fourieranalyse hat einen Speedup von 4,12 (= 443,51 / 107,5625)

Jan Niclas Ruppenthal
1481198

In den folgenden Tabellen befinden sich die Frequenzen und ihre Amplitudenmittelwerte, die größer sind als der übergebene Schwellwert, jeder Variante in jedem Experiment:

test01_sine.wav					
Sequenziell		Parallel mit Multiprocessing		Parallel mit OpenCL	
Frequenz	Amplitudenmittelwert	Frequenz	Amplitudenmittelwert	Frequenz	Amplitudenmittelwert
0.0	222.294.164	0.0	222.294.164	0.0	222202.27
86.133	242.419.295	86.133	242.419.295	86.133	242369.4
172.266	300.789.047	172.266	300.789.047	172.266	300766.84
258.398	430.139.419	258.398	430.139.419	258.398	430266.6
344.531	807.514.951	344.531	807.514.951	344.531	808773.44
430.664	8.227.258.524	430.664	8.227.258.524	430.664	8299535.0
516.797	1.001.750.106	516.797	1.001.750.106	516.797	998474.2
602.93	474.308.027	602.93	474.308.027	602.93	474086.62
689.062	312.171.274	689.062	312.171.274	689.062	312095.25
775.195	233552.66	775.195	233552.66	775.195	233594.88

test02_harmonics.wav					
Sequenziell		Parallel mit Multiprocessing		Parallel mit OpenCL	
Frequenz	Amplitudenmittelwert	Frequenz	Amplitudenmittelwert	Frequenz	Amplitudenmittelwert
0.0	234.268.196	0.0	234.268.196	0.0	234189.03
86.133	249.870.614	86.133	249.870.614	86.133	249874.83
172.266	293.115.469	172.266	293.115.469	172.266	293008.5
258.398	379.226.349	258.398	379.226.349	258.398	379574.88
344.531	571.137.126	344.531	571.137.126	344.531	571620.9
430.664	1.256.082.314	430.664	1.256.082.314	430.664	1257164.4
516.797	5.157.233.863	516.797	5.157.233.863	516.797	5155579.5
602.93	859.680.084	602.93	859.680.084	602.93	858094.1
689.062	508.932.056	689.062	508.932.056	689.062	509434.6
775.195	437.735.027	775.195	437.735.027	775.195	438015.2
861.328	562.742.731	861.328	562.742.731	861.328	562620.6
947.461	1364220.15	947.461	1364220.15	947.461	1362142.1
1.033.594	2.117.790.589	1.033.594	2.117.790.589	1.033.594	2116476.8
1.119.727	622.986.308	1.119.727	622.986.308	1.119.727	622992.1
1.205.859	403.385.687	1.205.859	403.385.687	1.205.859	403658.72
1.291.992	352.609.553	1.291.992	352.609.553	1.291.992	352581.34
1.378.125	440.591.559	1.378.125	440.591.559	1.378.125	440891.44
1.464.258	1.363.830.496	1.464.258	1.363.830.496	1.464.258	1362987.9
1.550.391	971.827.185	1.550.391	971.827.185	1.550.391	971173.8
1.636.523	382.327.446	1.636.523	382.327.446	1.636.523	382173.2
1.722.656	263.388.298	1.722.656	263.388.298	1.722.656	263341.56
1.808.789	231.796.967	1.808.789	231.796.967	1.808.789	231494.55
1.894.922	276.688.835	1.894.922	276.688.835	1.894.922	276624.56
1.981.055	1.270.724.376	1.981.055	1.270.724.376	1.981.055	1271255.9
2.067.188	367476.88	2.067.188	367476.88	2.067.188	367931.44

test03_add.wav					
Sequenziell		Parallel mit Multiprocessing		Parallel mit OpenCL	
Frequenz	Amplitudenmittelwert	Frequenz	Amplitudenmittelwert	Frequenz	Amplitudenmittelwert
0.0	347.998.807	0.0	347.998.807	0.0	345270.66
86.133	441.929.775	86.133	441.929.775	86.133	439725.28
172.266	723.647.951	172.266	723.647.951	172.266	728979.2
258.398	2.210.815.426	258.398	2.210.815.426	258.398	2239768.8
344.531	2.065.454.869	344.531	2.065.454.869	344.531	2028802.1
430.664	707.846.607	430.664	707.846.607	430.664	706487.44
516.797	453.116.385	516.797	453.116.385	516.797	454021.25
602.93	3.352.824.372	602.93	3.352.824.372	602.93	3368083.2
689.062	475.202.701	689.062	475.202.701	689.062	470889.8
775.195	745.348.264	775.195	745.348.264	775.195	743044.7
861.328	2.354.183.652	861.328	2.354.183.652	861.328	2327354.5
947.461	1.918.807.353	947.461	1.918.807.352	947.461	1932248.8
1.033.594	689.002.098	1.033.594	689.002.098	1.033.594	686286.0
1.119.727	425.491.349	1.119.727	425.491.349	1.119.727	422554.62
1.205.859	311.122.279	1.205.859	311.122.279	1.205.859	310516.97
1.291.992	247.244.601	1.291.992	247.244.601	1.291.992	245570.16
1.378.125	206.418.824	1.378.125	206.418.824	1.378.125	206579.67

test04_segmented.wav					
Sequenziell		Parallel mit Multiprocessing		Parallel mit OpenCL	
Frequenz	Amplitudenmittelwert	Frequenz	Amplitudenmittelwert	Frequenz	Amplitudenmittelwert
0.0	473.222.727	0.0	473.222.727	0.0	474845.47
86.133	513.194.544	86.133	513.194.544	86.133	512925.47
172.266	631.883.681	172.266	631.883.681	172.266	632719.1
258.398	916.990.974	258.398	916.990.974	258.398	910706.7
344.531	2.073.644.243	344.531	2.073.644.243	344.531	2084627.6
430.664	3.624.343.849	430.664	3.624.343.849	430.664	3540353.8
516.797	1.212.098.646	516.797	1.212.098.646	516.797	1217781.5
602.93	975980.24	602.93	975980.24	602.93	974992.94
689.062	1.180.282.119	689.062	1.180.282.119	689.062	1170640.9
775.195	3.927.925.996	775.195	3.927.925.996	775.195	3890458.8
861.328	1.706.552.734	861.328	1.706.552.734	861.328	1739264.2
947.461	811.097.065	947.461	811.097.065	947.461	808267.6
1.033.594	560.032.326	1.033.594	560.032.326	1.033.594	560790.06
1.119.727	436.745.702	1.119.727	436.745.702	1.119.727	437918.25
1.205.859	361748.71	1.205.859	361748.71	1.205.859	363363.16
1.291.992	310.637.487	1.291.992	310.637.487	1.291.992	307863.03
1.378.125	273.242.071	1.378.125	273.242.071	1.378.125	275401.12
1.464.258	244.535.159	1.464.258	244.535.159	1.464.258	243650.97
1.550.391	221.714.133	1.550.391	221.714.133	1.550.391	222650.47
1.636.523	203.083.859	1.636.523	203.083.859	1.636.523	201562.3

Wenn man die Amplitudenmittelwerte der parallelen Varianten mit OpenCL betrachtet, dann erkennt man, dass die Resultate eine Abweichung gegenüber den Mittelwerten der anderen beiden Implementierungen haben. Hierbei ist der Grund, dass für OpenCL nicht die gleiche Implementierung der Fourieranalyse verwendet werden konnte.

Diskussion der erreichten Ergebnisse

Die parallele Variante der FFT-Analyse erreicht **keinen** Speedup von 16. Dies liegt auch daran, dass für die Addition der Zwischenergebnisse alle Threads synchronisiert, werden müssen, damit keine Race Conditions entstehen.

Des Weiteren erreicht auch die parallele Implementierung für die GPU auch **nicht** einen Speedup von 8. Dies liegt vor allem daran, dass die Daten in jeder Iteration auf den Speicher der GPU geschrieben werden müssen. Hierbei wird auch ein Puffer für die jeweiligen Variablen erstellt. Die Zwischenergebnisse werden in jedem Durchlauf von der CPU in die Variable **aggregated_fft** addiert. Des Weiteren wurde auch ein einfacher Algorithmus der Fouriertransformation implementiert. Dies lag aber daran, dass bei jeder anderen Implementierung Schwierigkeiten auftraten, wie bspw. das Einfrieren des gesamten Betriebssystems oder es konnten nicht genug Ressourcen zur Verfügung gestellt werden, auch wenn eine kleine Audiodatei übergeben wurde.

Des Weiteren hätte man auch eine andere Programmiersprache wählen können. Hierbei wären Sprachen wie Java oder Rust eine bessere Wahl gewesen.