Heterogeneous Computing, Übungsblatt 2, Sommer 2024

Implementierung der Aufgaben:

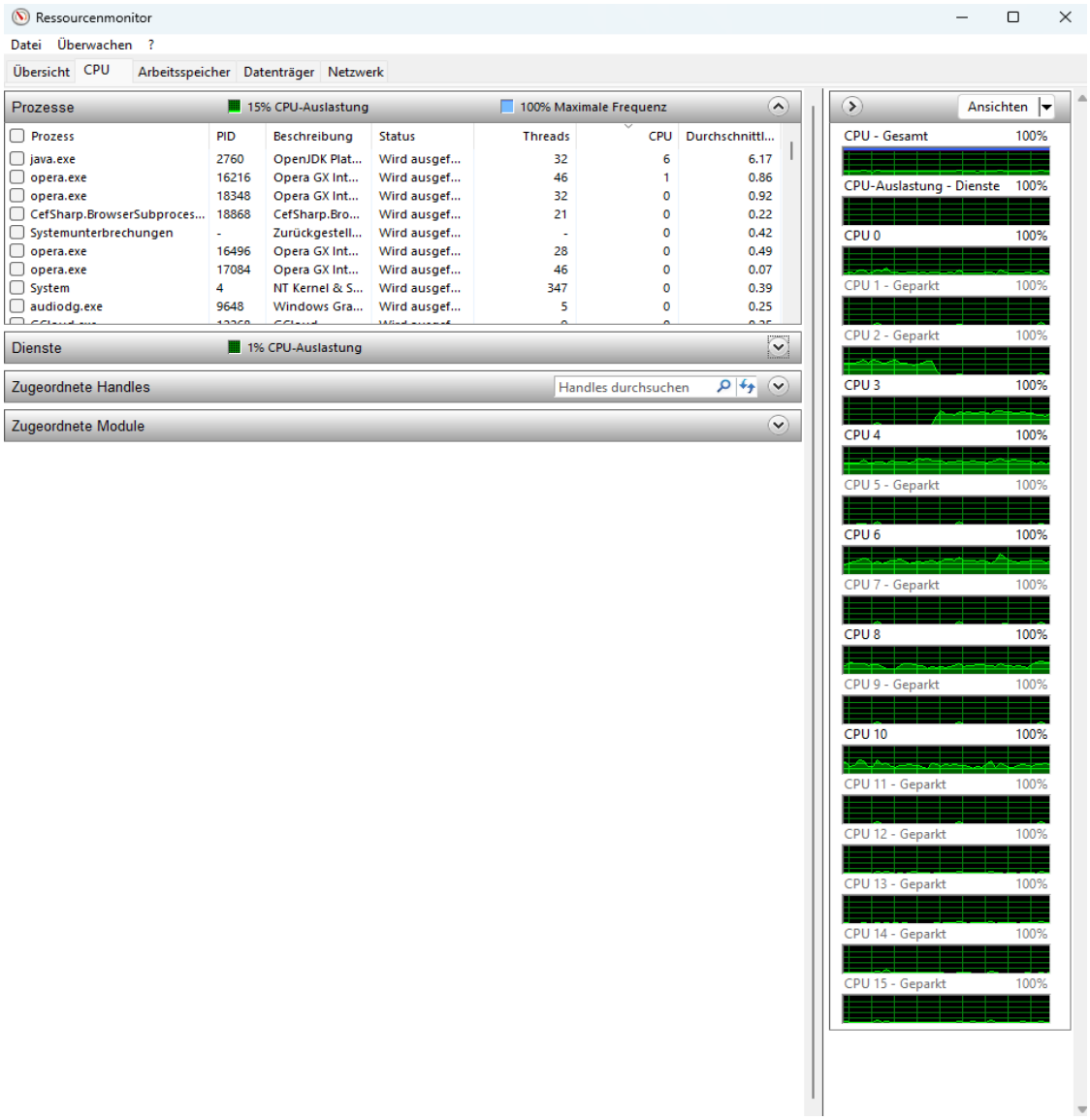
Aufgabe 1: Sequenzielle Lösung

Im Gegensatz zu Übungsblatt 1 habe ich bei dieser Implementierung Java verwendet, was direkt zu einer weitaus besseren Performance führte als mit Python.

Wie Gewünscht können die Parameter Blockgröße, Versatz, Schwellenwert und Ort der Datei von dem Nutzer frei gewählt werden.

Aufgrund der sequenziellen Implementierung ist das Programm ziemlich langsam und und benötigte für die Haupttestdatei 5 min.

Beispiel zur CPU-Auslastung:

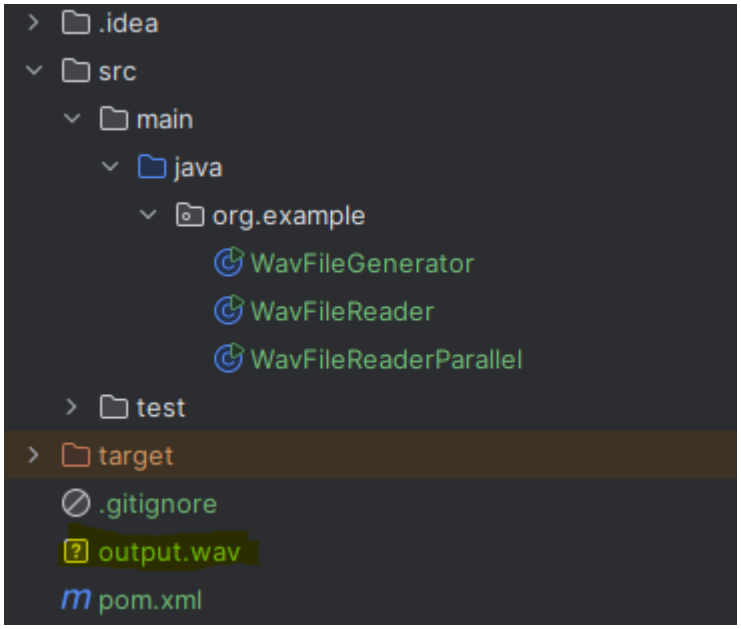


Beispiel Ausgabe:

Aufgabe 2: Generierung von WAV-Testdateien

Die WAV-Dateien können bei meinem Code mit beliebigen Kanälen, Frequenzen sowie Dauer erstellt werden.

Die ausgegebene Datei erhält standardmäßig den Namen “output.wav” und wird im Arbeitsverzeichnis abgespeichert.



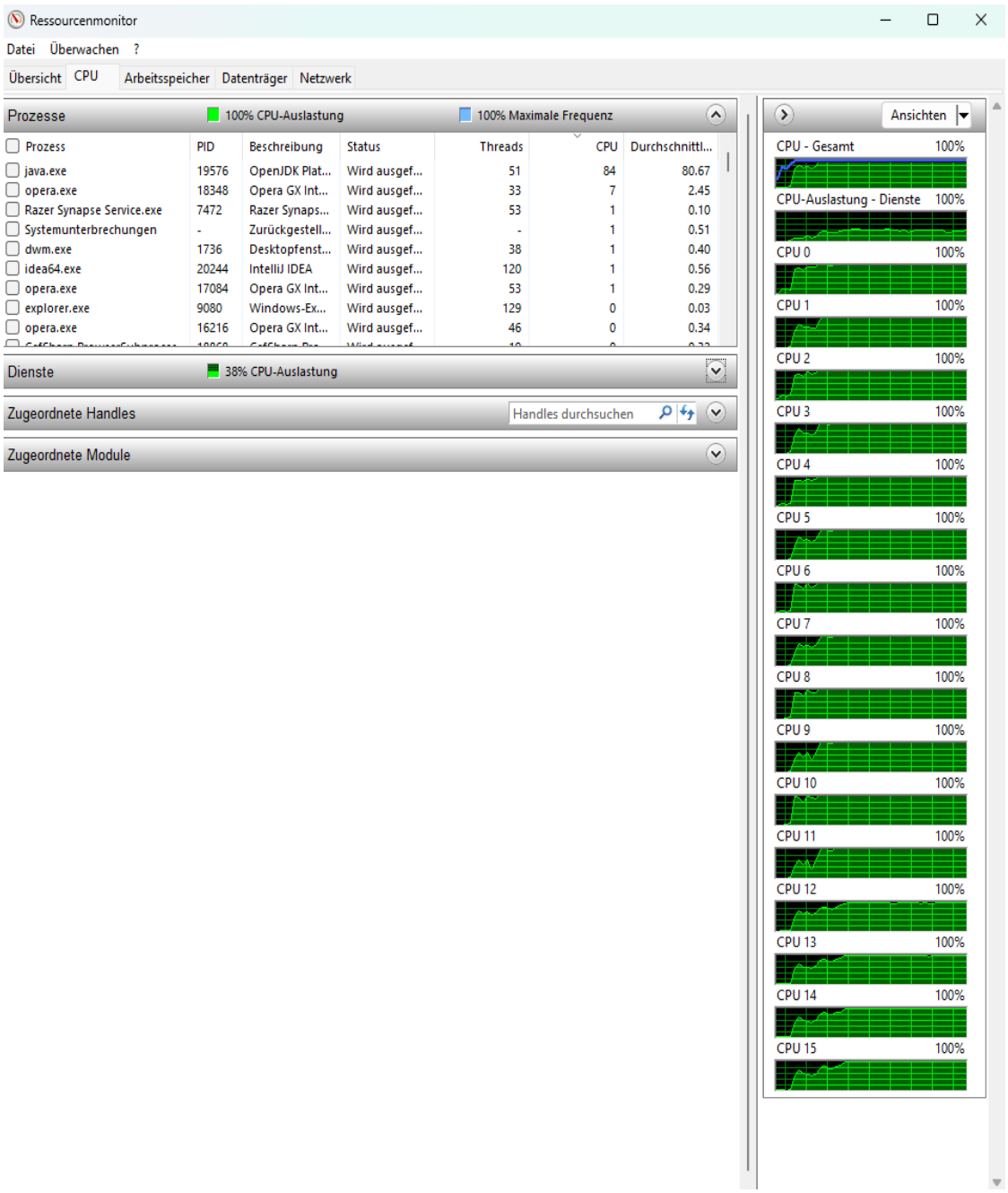
Aufgabe 3: Parallele Lösung (CPU)

Die parallele Lösung nutzt den ExecutorService zur parallelen Verarbeitung.

Dieser erstellt die gewünschte Anzahl an Threads und arbeitet diese ab, bis alle Aufgaben abgeschlossen wurden.

Im Vergleich zur sequenziellen Lösung wurden alle Kerne der CPU so gut wie dauerhaft vollständig von dem Programm ausgenutzt. Dies führte zu einer Verbesserung des Programms um das 5-fache.

Beispiel CPU:



Beispiel Ausgabe:

Frequenz: 21619.3359375 Hz, Durchschnittliche Amplitude: 0.22702710804781268

Frequenz: 21834.66796875 Hz, Durchschnittliche Amplitude: 0.2442459559698021

Frequenz: 215.33203125 Hz, Durchschnittliche Amplitude: 0.2442467387020175

Frequenz: 430.6640625 Hz, Durchschnittliche Amplitude: 0.22702620743044716

Programm dauerte 59.7823467 Sekunden.

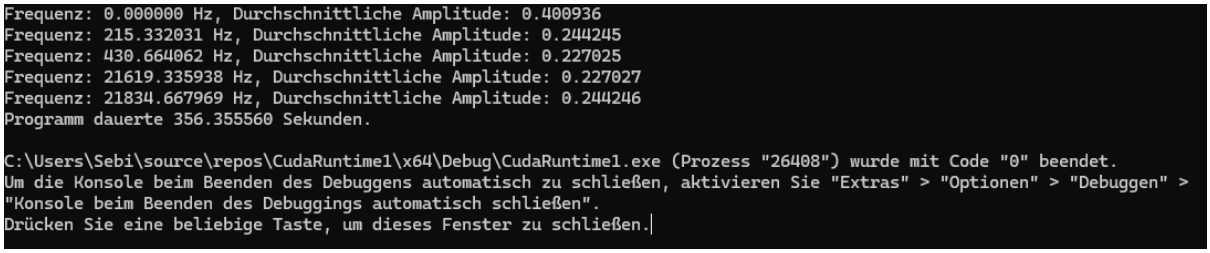
Aufgabe 4: Parallele Lösung (GPU)

Bei der parallelen Lösung für die GPU habe ich CUDA genutzt, da ich eine NVIDIA-Grafikkarte besitze.

Die Parameter sind gleich zu den vorherigen Lösungen mit der Ausnahme der BLOCK_SIZE, die bestimmt die Anzahl der Threads in einem Block für die CUDA-Kernel-Funktionen.

Erstaunlicherweise habe ich es nicht geschafft den Cuda Code nicht wesentlich effizienter Laufen zu lassen als meinen parallelen Code in Java.

Beispiel Ausgabe:



Test mit den Programmen:

“Standard”-Test:

Die Ergebnisse für ein WAV-Datei mit 60 Sekunden können Sie bereits oben sehen.

Hier benötigte die sequenzielle Lösung 5 Minuten, die parallele Lösung 60 Sekunden und 6 Minuten für die parallele GUI-Lösung.

Kurze WAV-Datei:

Nutzt man eine sehr kurze WAV-Datei (5 Sekunden) so benötigen die Programme folgend lang:

Sequenziell: 24 Sekunden

Parallel CPU: 6 Sekunden

Parallel GPU: 30 Sekunden

Dieser Test zeigt, dass selbst bei kleinen WAV-Dateien der Overhead der parallelen Berechnung weitaus niedriger ist, als die Zeitersparnis durch die multiplen Threads.

Threads:

Beim Ändern der Anzahl der Threads ergab sich bei meinem Code keinen großen Unterschied bei der parallelen CPU. Getestet habe ich mit 4,8,16 und 32 Threads.

Bei der parallelen Verarbeitung mit der GPU ergaben sich jedoch größere Unterschiede von mehreren Sekunden (+5 bis -10). Bei der Anpassung von 1024 auf 512 bzw. 2048 BLOCK_SIZE.

Erhöhung der Frequenzen:

Verdoppelt man die Anzahl der genutzten Frequenzen, so erhöht sich auch die Zeit extrem, wobei die Vervielfachung zwischen den Methoden unterschiedlich war.

Sequenzielle CPU: 300 auf 1167 Sekunden

Parallele CPU von 60 auf 89 Sekunden