Heterogeneous Computing
Ergebnisbericht

Tillmann Faust

14.7.2024

Aufgabe 1

Zum Lösen der ersten Aufgabe, wurde der Aufgabentext mithilfe von ChatGPT 4o realisiert. Das Programm geht das data Array mit Verschiebungen der Größe 'offset' sequentiell durch, berechnet für die Blöcke der Größe block_size die Amplituden, summiert diese währenddessen auf und berechnet anschließend den Schnitt. Anschließend werden alle Frequenzen bestimmt, alle Frequenzen mit Amplituden über 'threshold' bestimmt und ausgegeben. Das Ergebnis ist das Programm "fft single_thread.py".

Aufgabe 2

Das Programm "generate_wav_files.py" generiert vier wav Dateien, welche -rspektiv -aus einer einzelnen 500Hz Sinus-Welle, einer Mischung von drei Sinuswellen unterschiedlicher Frequenz, Sinuswellen mit steigender Amplitude, und White Noise bestehen. Die Dateien wurden so gewählt um zu beobachten, ob die Zusammensetzung des Eingabesignals einen Einfluss auf die Laufzeit hat. Die vorherige Überlegung, dass dies nicht der Fall sein wird wurde allerdings bestätigt. Interessant wäre eventuell noch gewesen zu beobachten, wie sich die Laufzeit bei einem Vielfachen der Datenmenge verändert.

Aufgabe 3

Die Grundidee des Programms ist, den Datenstrom blockweise aufzuteilen, also die Menge der zu bearbeitenden Blöcke in num_threads gleich große Gruppen zu unterteilen und jede Gruppe von einem eigenen Thread bearbeiten zu lassen. Ursprünglich wurde dazu das Modul 'multithreading' verwendet, dies führte allerdings zu dem Problem, dass es zu keiner wirklichen Paralellisierung kam, weswegen sich anschließend für das Modul 'multiprocessing' entschieden wurde. Die erste Version des Programms berechnet die zu bearbeitenden Blöcke im sequentiellen Teil vor, übergibt die einzelnen Blocklisten an die Prozesse, welche anschließend die FFT durchführen. Die Probleme die dabei entstehen sind allerdings zum einen, dass der Sequentielle Teil erheblich länger dauert und

zum Anderen entsteht ein großer Speicherplatzverbrauch.

Bei einer Dateigröße von ca. 3GB umfasst das 'data' Array 441.000.000 Datenpunkte, womit die Zahl der zu berechnenden Blöcke 27.562.468 beträgt. Die zweite Variante des Programms, also die Abgabe, berechnet lediglich die Start- und End-Indizies der 'data'-Array Abschnitte, welche jeder Thread zu berechnen hat und die Blöcke werden in jedem Thread bestimmt und abgearbeitet. Dadurch wird das Berechnen der tatsächlichen Blöcke in den parallelen Teil verschoben und der Speicherverbrauch wird bedeutend reduziert. Das Ergebnis ist das Programm "fft multi_thread cpu.py".

Aufgabe 4

Zur Umsetzung der 4. Aufgabe müssen zuerst Vorüberlegungen bezüglich der Aufgabenverteilung und Datenverwaltung gemacht werden um die verfügbaren Ressourcen möglichst effizient nutzen zu können. Es muss allerdings leider direkt angemerkt werden, dass das Programm leider kein Brauchbares Ergebnis liefert, sondern als Ergebnis lediglich ein Array aus block_size/2 vielen NaN-Werten besteht. Zur GPU-Programmierung selbst wurde OpenCL über pyopencl verwendet.

Vorüberlegungen

1. Aufgabenverteilung Zu beginn des Programms wird die Zahl der verwendeten Kerne fest und das Kernel Programm ist so geschrieben, dass sich ein Kern so lange neue Aufgaben holt, bis die Berechnung Aller Blöcke abgeschlossen sind. Die Aufgabenverteilung erfolgt über einen task_counter welcher synchronisiert hochgezählt wird, bis alle Blöcke berechnet wurden.

2. Datenverwaltung Es wird ein Array mit genau block_size/2 vielen Plätzen initialisiert auf welches die Kerne synchronisiert zugreifen können um dort die einzelnen Werte zu akkumulieren. Die zweite Variante birgt allerdings das Problem, dass Kerne einander stärker einschränken, da das Schrieben der Ergebnisse immer nur von einem Kern zeitgleich vorgenommen werden kann.

Umsetzung

Die Umsetzung des Kernel-Programms und die Verwaltung der Kerne und Daten wurden bereits in den Vorüberlegungen besprochen. Leider liefert Programm nicht die erhofften Ergebnisswerte. Untersuchungen haben ergeben, dass in den FFT-Berechnungen der einzelnen Blöcke teilweise so hohe Werte entstehen, dass der Ergebniswert als 'inf' gespeichert wird. Folgeberechnungen führen zu der Ausgabe der NaN Werte. Ich vermute, dass dies an einem fehlerhaften FFT-Algorithmus liegt und konnte bis zur Abgabe leider nicht korrigiert werden. Das umschließende Python Programm verarbeitet die Daten vor, indem Variablen in die richtigen Typen umgewandelt, Puffer Strukturen angelegt und die für

die Übergabe der Aufgaben an die GPU notwendigen Strukturen initialisiert werden.

Analyse

Details

Die Programmtests für ”fft single_thread.py” und ”fft multi_thread cpu.py” wurden auf einem Microsoft Laptop Studio 2 durchgeführt. CPU: Intel Core i7-13700H @ 2,90 GHz, GPU: NVIDIA GeForce RTX 4050 Laptop GPU. Die CPU besitzt 20 logische Kerne, entsprechend wurden für jeden Multithread Test 20 Threads verwendet. Die verwendeten Programmparameter waren

- wav_file: In Aufgabe 2 Generierte wav-Dateien
- block_size = 512
- offset = 16
- threshold = 50

Ergebnisse

Die Ergebnisse der Ausführungen sind in Fig. 1 graphisch dargestellt. Es ist ein klarer Speedup von knapp unter 10 zu erkennen, was der Hälfte der verwendeten Kerne entspricht. Es ließ sich beobachten, dass ein großer Teil der Laufzeit der Multithread Version für das Einlesen und Vorverarbeiten der wav-Dateien, sowie das erstellen der einzelnen Prozesse aufgewendet wird. Weiterhin ist Python als Sprache vergleichsweise langsam, weswegen die Sequentielle Teile einen zusätzlich höheren Zeitaufwand bergen.

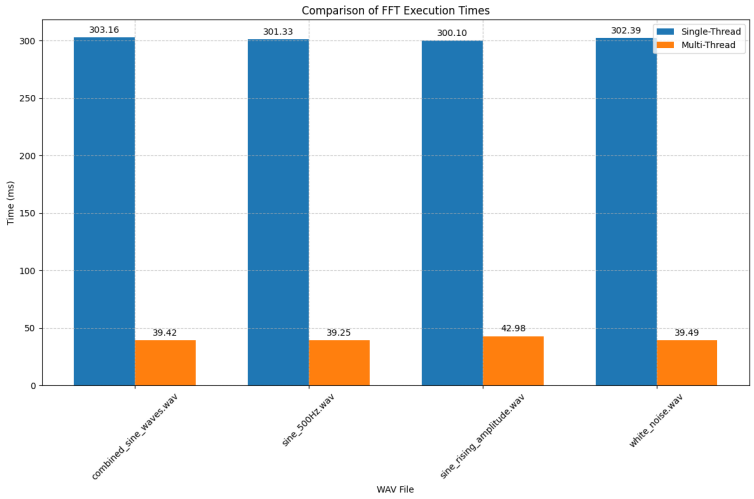

Figure 1: Laufzeitvergleich der Single- und SPU-Multi-Thread Lösungen

Da das Programm ”fft multi_thread gpu” bis zuletzt keine brauchbaren Werte liefern konnte, wurde es vom Vergleich ausgeschlossen, allerdings lag die Ausführunszeit des Programms über der des sequentiellen Programms, weswegen die Vermutung besteht, dass das Programm mehr Probleme aufweist, als die Mangelhafte Berechnung der FFT selbst.