

Heterogeneous Computing

Übungsblatt 2

Kilian Bartz (Mknr.: 1538561)

Hinweis. Der zugehörige Code befindet sich im folgenden Github Repository: https://github.com/kilianbartz/hetcomp_ueb02.

Aufgabe 1

Zur Implementierung von Aufgabe 1 und Aufgabe 3 entschied ich mich dazu, Rust als systemnahe und sehr performante Programmiersprache zu verwenden. Neben der Performanz war vor allem die Programmiererfahrung durch moderne Tools (rust Compiler, cargo Package Manager) sehr angenehm. Als Libraries kommen *argparse* (zum Parsen der CLI Argumente), *hound* (zum Lesen der WAV-Dateien), *itertools.num* (zum Berechnen der beteiligten Frequenzen) und *rustfft* als schnelle Implementierung der FFT zum Einsatz. Diese Library verwendet bei unterstützter Hardware AVX-512 Vektorisierung, sodass 16 Single-Precision Float Operationen parallel (auf einem Prozessorkern) ausgeführt werden könnten. Dadurch kommt es zu paralleler Datenverarbeitung, während immer noch nur sequenzielle Instruktionen ausgeführt werden (analog dazu, wie ein bitweises Oder auf allen Bits der Eingaben gleichzeitig arbeitet).

Die Implementierung folgt meiner ursprünglichen Python Lösung für Übung 1: Die Audiodatei wird eingelesen und bei Bedarf von Stereo in Mono umgewandelt (dabei wird der Einfachheit halber nur der erste Audiokanal benutzt; alternativ könnten die Signale beider Kanäle gemittelt werden). Anschließend wird auf jedem Block die FFT Berechnung durchgeführt, die Amplituden in dB berechnet und die Hauptfrequenzen in einem Vektor **stats** gesammelt (hierbei werden aus Effizienzgründen zwei Sortiervorgänge der Pythonlösung eingespart). Abschließend werden die Statistiken in eine Datei geschrieben. Durch das Sammeln aller Statistiken und das einmalige Schreiben werden IO-Bottlenecks vermieden, jedoch steigt dadurch natürlich der Arbeitsspeicherbedarf.

Aufgabe 2

Zur Generierung von WAV-Testdateien kommt ein Python Skript zum Einsatz. Es werden die selbsterklärenden CLI Argumente *duration*, *output path*, *wave type*, *amplitude*, *sample-rate* und *frequencies* eingelesen, um eine entsprechende Mono WAV-Datei zu generieren. Der Wellentyp der generierten Signale hat keinen Einfluss die Dauer der Fourier-Transformation, da die Wellen sich ohnehin überlagern. Relevant für die Experimente sind daher nur die Parameter, welche die letztendliche Anzahl der Samples beeinflusst, also *duration* und *sampling-rate*.

Aufgabe 3

Zur Parallelisierung des Codes von Aufgabe 1 verwendet diese Implementierung die Rust Library *rayon*¹. Diese bietet ein sehr simples Interface (`x.par_iter()`, `iter.into_par_iter()`), um Berechnungen auf einem Iterator, wie die hier durchgeführte Map-Operation der Audiosamples aus einem Block auf die enthaltenen Hauptfrequenzen, parallel auszuführen. Dadurch lassen sich diese Berechnungen durch das Hinzufügen einer einzigen Code Zeile auf alle CPU-Kerne verteilen. Dabei garantiert die Library, dass keine Data-Races auftreten können.

¹<https://github.com/rayon-rs/rayon>

Aufgabe 4

Zur Parallelisierung auf der GPU verwende ich AMDs HIP SDK for Windows². Während es für Linux mit AMD ROCm³ einen Softwarestack gibt, der mit dem Funktionsumfang des CUDA Toolkits vergleichbar ist, stellt das HIP SDK leider nur einen sehr eingeschränkten Teil der ROCm Tools für Windows bereit. Enthalten sind zwar die für diese Aufgabe benötigten Mathematik-Libraries (rocFFT/hipFFT) und ein Compiler, jedoch weder ROCgdb als Debugger oder rocprofiler. Zudem ist die Dokumentation noch sehr spärlich.

Hinsichtlich der Libraries gibt es für viele Anwendungen die Wahl zwischen einer “hip”- und einer “roc”-Version. Dabei stellen die “roc”-Versionen Open-Source Libraries für AMD Hardware dar und unterscheiden sich im Interface stärker von den CUDA-Pendants. Demgegenüber bieten die “hip”-Versionen ein CUDA-ähnliches Interface und erlauben es, denselben Code sowohl für AMD-Hardware als auch für Nvidia-Hardware zu kompilieren (hipFFT verwendet beispielsweise dazu rocFFT für AMD bzw. cuFFT für Nvidia). Aufgrund der höheren Flexibilität und der angenehmeren API entschied ich mich für diese Aufgabe für hipFFT.

hipFFT⁴ bietet als hardwarenahe FFT Implementierung viele Konfigurationsoptionen, z. B. Art (vorwärts, rückwärts), Typ (real \rightarrow komplex, komplex \rightarrow real, komplex \rightarrow komplex), Speicherverwaltung (in-place, out-of-place), Speicherformat, usw. Der generelle Ablauf ist dabei an FFTW als populärste FFT Library für C angelehnt: Zunächst wird ein FFT-Plan angelegt, der basierend auf der Hardware und der Konfiguration einen Algorithmus auswählt und die FFT-Berechnungen optimiert. Nach dem Allokieren der Input-/Outputbuffer werden die Daten (Audiosamples) auf die GPU kopiert und der FFT-Plan wird ausgeführt. Dies kann weiter durch Batch-Aufrufe optimiert werden, sodass mehrere Blöcke (nebeneinander im Input-Buffer) parallel verarbeitet werden. Anschließend kann der Outputbuffer zurück auf den Host kopiert oder weiterverarbeitet werden.

Meine Implementierung

Nach dem Parsen der CLI Argumente und dem Einlesen der Audiodatei (wieder nur erster Audiokanal) werden die Anzahl der zu berechnenden Batches ausgerechnet und entsprechende FFT-Pläne für normale Batches und die eventuell auftretende letzte kleinere Batch angelegt. Zusätzlich werden die Frequenzen, zu denen die in der FFT berechneten Amplituden gehören, bereits vorab auf der CPU berechnet, da diese konstant bleiben.

Innerhalb jeder Iteration der Hauptschleife über alle Batches wird nun ein Vector mit den Blöcken für die FFT befüllt. Dies ist nicht nur notwendig, da hipFFT erwartet, dass im Inputbuffer alle Blöcke nebeneinander stehen, sondern auch, da hipFFT auch bei separatem Outputbuffer Teile des Inputbuffers während der Berechnung überschreibt. Nachdem die FFT berechnet wurde, transformiert ein simpler Kernel (`convert_to_dB`) die komplexen FFT-Ergebnisse in dB Amplituden. Im Anschluss daran werden die Daten erst wieder zurück auf den Host kopiert und in einem Vector gesammelt. Wie auch in den anderen beiden Implementierungen (Aufgabe 1, Aufgabe 3) werden die Statistiken erst nach dem Beenden aller Berechnungen in eine Datei geschrieben.

Hinweis.

Theoretisch ließe sich meine HIP Anwendung durch größere Batchgrößen sicherlich noch weiter optimieren, da dadurch die Zahl der Kopiervorgänge Host \rightarrow GPU und GPU \rightarrow Host reduziert werden würde (ein Verdoppeln von 500 auf 1000 verringert die benötigte Zeit etwa um 20%). Jedoch scheint `max_batch_size = 1000` ein nicht dokumentiertes Limit zu sein; wenn ein Wert > 1000 verwendet wird, sind die Ergebnisse der FFT nicht mehr korrekt, obwohl es zu keiner Fehlermeldung kommt und auch der Speicher der GPU nicht voll läuft. Dies war in meinen Experimenten auch unabhängig von der Blockgröße der Fall. Bei cuFFT gibt es ein Limit von 2^{27} Elemente im Inputbuffer, die hier jedoch nicht ansatzweise erreicht werden. Ohne Debugger und Profiler ist es mir jedoch leider nicht möglich, dieses Problem weiter zu untersuchen.

²<https://www.amd.com/en/developer/resources/rocm-hub/hip-sdk.html>

³<https://rocm.docs.amd.com/en/latest/>

⁴<https://rocm.docs.amd.com/projects/hipFFT/en/latest/api.html>

Experimente

Als Testsystem kommt ein Windows PC mit einem AMD Ryzen 7 7700X (8C/16T; unterstützt AVX-512), 32 GB RAM und einer AMD RX 6900XT (5120 Shader) als GPU zum Einsatz. Damit eine Laufzeit der sequenziellen Lösung für die Baseline in 7:18min erreicht wird, werden folgende Parameter (sofern nicht anders genannt) verwendet:

- Blockgröße: 16384
- Schrittweite/Versatz: 1
- Schwellwert: 50dB
- Dauer der Audiodateien: 3min
- Sampling-Rate: 44100 Hz

Folgende Experimente wurden durchgeführt:

1. Sinuswellen mit 440, 880, 1200, 2400Hz als Baseline
2. analog zu Experiment 1 mit Blockgröße 4096 ($\frac{1}{4}$ der Baseline) und Länge der Audiodatei von 12min (4x Baseline)
3. analog zu Experiment 1 mit Blockgröße 32768 (Scaling mit großen Blockgrößen)
4. analog zu Experiment 1 mit Blockgröße 128 (Scaling mit kleinen Blockgrößen)
5. Wiederholung des Experiments aus Übungsblatt 1: Audiodatei "nicht_zu_laut_abspielen.wav", Blockgröße 2055

	Experiment 1	Experiment 2	Experiment 3	Experiment 4	Experiment 5
Sequenziell	435	421	1013	5,3	83
Multicore	52	51	113,6	0,5	10
GPU	171	188	415	6	19

Tabelle 1: Laufzeiten (in sek) der verschiedenen Implementierungen in den Experimenten 1 - 5. Es handelt sich dabei um die reine Rechenzeit exklusive Schreiben der Ergebnisse

Die Ergebnisse finden sich in Tabelle 1 und Abbildung 1. Alle Implementierungen produzieren identische, korrekte Ergebnisse mit ausschließlich hohen Amplituden um die entsprechenden Frequenzen.

Mit Blick auf Experiment 5 sieht man, dass meine neue sequentielle Lösung aus Aufgabe 1 durch rust, AVX und kleine Codeoptimierungen nur noch ca. 83s statt 400s (ursprüngliches Python Script) benötigt und damit einen Speedup von 8.4 erreicht.

Generell liefert die Multi-Core Lösung mit Abstand die beste Performance mit einem durchschnittlichen Speedup von etwa 8.6. Dies zeigt eindrucksvoll die Möglichkeiten, die rust und rayon zur effizienten parallelen Datenverarbeitung bieten. Dass der Speedup etwas größer als die Anzahl der physischen Kerne ist, zeigt, dass Hyperthreading (bzw. SMT) durchaus durch bessere Ressourcenverteilung die Effizienz verbessern. Dies ist vor allem vor dem Hintergrund interessant, dass der Prozessor beim Ausführen der Multicore-Lösung trotz Wasserkühlung aufgrund der großen Last (AVX-512 Code auf allen Kernen) um etwa 4.5% herunter taktet. Die GPU Lösung erreicht einen mittleren Speedup von ca. 2.45 gegenüber der sequenziellen Lösung; die Multicore-Lösung läuft jedoch im Schnitt noch einmal 3.51 mal schneller. Dies liegt höchstwahrscheinlich an der relativ kleinen `max_batch_size` in der GPU-Lösung und dem damit einhergehenden ständigen Kopieren zwischen Host und GPU. Deshalb ist die GPU laut Task-Manager auch nur zu zwischen 25 – 34% ausgelastet und es werden nur 3.4/16 GB VRAM verwendet. Bei voller Auslastung wäre es denkbar, dass die GPU Lösung zumindest nahe an die Multicore-Lösung herankommt.

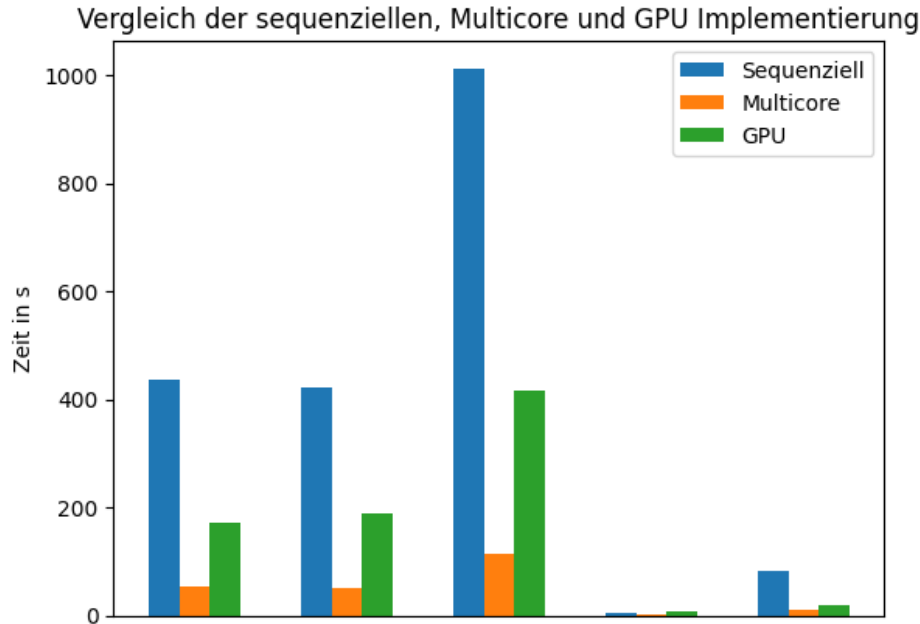


Abbildung 1: Plot der Laufzeiten aus Tabelle 1

Trotz der ineffizienten Nutzung der GPU sieht man in Experiment 5, dass der Kernel, der von *hipFFT* für kleine Blockgrößen verwendet wird, welche keine 2er Potenzen sind, relativ gesehen effizienter sind als der Algorithmus von *rustfft*, da der Speedup hier auf 1.9 schrumpft.

Selbst bei sehr kleinen Blockgrößen und entsprechend niedrigem Rechenaufwand (Experiment 4) bleibt der Speedup der Multicore-Lösung erhalten, was den sehr geringen Multicore-Overhead von *rayon* hervorhebt. Da hier in der GPU-Lösung trotz der kleinen Blockgröße immer noch nur 1000 Blöcke auf einmal bearbeitet werden können, wird der Overhead so groß, dass die GPU-Lösung nun sogar schlechter wird als die sequenzielle.

Auch Experiment 2 zeigt, dass die GPU-Lösung größere Blockgrößen favorisiert, da der gesamte Rechenaufwand trotz geringerer Blockgröße sehr ähnlich ist wie in Experiment 1, die Laufzeit der GPU-Lösung jedoch um 10% steigt, während die Laufzeit der anderen Lösungen um ca. 2 – 3% sinkt.

Insgesamt erreicht meine GPU-Lösung nie einen Speedup von k (in diesem Fall wäre $k = 5120$ die Zahl der Shader-Einheiten), was auch aufgrund der mangelnden Leistungsfähigkeit dieser Einheiten im Vergleich mit einem CPU-Kern, dem Fehlen von AVX und der deutlich geringeren Taktung von etwa 1.7 GHz gegenüber 5.15 GHz der CPU eine sehr optimistische Annahme wäre. In Anwendungen, in denen die GPU jedoch vollständig benutzt wird (Rendern von Videospielen, neurale Netzwerke) und ihre schiere Anzahl an Recheneinheiten effektiv nutzen kann, erreicht eine normale CPU auch mit AVX nicht ansatzweise eine vergleichbare Leistung.