Supersonic Algorithms

Anton Rodenwald

11. Januar 2023

Projektbetreuerin:	Birgit Ziegenmeyer
Institution:	Schillerschule Hannover
Thema des Projektes:	Analyse verschiedener Implementationen von Algorithmen in Bezug auf die besten Techniken zur Optimierung und Auseinandersetzung mit der Ausführung dieser, um die Umsetzung hochperformanter und effizienter Programme zu erforschen.
Fachgebiet:	Mathematik/Informatik
Wettbewerbssparte:	Jugend Forscht
Bundesland:	Niedersachsen
Wettbewerbsjahr:	2023

Kurzfassung

Nachdem wir im Informatikunterricht der SEK II Sortieralgorithmen behandelt hatten, stellte ich mir die Frage, wie man am schnellsten eine Liste von 10 Millionen zufällig Generierten Zahlen sortieren kann und welche Programmiersprache und welche Techniken man nutzen sollte. Daraus entwickelte sich dann die etwas allgemeinere Fragestellung, nämlich welche Optimierungen erhöhen die Ausführgeschwindigkeit von Programmen am meisten und wieso? Mir war bekannt, dass Python, was wir im Unterricht verwendet hatten, als eine der langsamsten Sprachen gilt, weswegen ich neben Python auch noch C++ wählte, was allgemein als eine der schnellsten Sprachen gilt. Ich implementierte anschließend verschiedene Variationen der Quicksort und anderer Algorithmen und testete so, in welchem Maß Optimierungsansätze die Performance beeinflussten. Dabei kam ich zu dem Ergebnis, dass die besten Python Bibliotheken zur Optimierung numpyünd numba"waren, wobei C++ trotzdem schneller war, womit sich meine Hypothese bestätigte. Dies erklärte ich mir dadurch, dass Python eine Interpretierte und C++ eine kompilierte Sprache ist, diese beiden Sprachen also gänzlich verschiedenen sind und somit auch die Möglichkeiten zur Optimierung dieser Andere sind. Schlussendlich gelang es mir noch unter Nutzung von AVX2, einem speziellen Befehlssatz, in C++ eine 4x schnellere Version als die standardmäßig Vorhande zu entwickeln was mir zeigte, dass es im Gebiet der Codeoptimierung noch viel zu entdecken und zu testen gibt. Im breiteren Kontext gesehen, sind Optimierungen hilfreich, um die Ausführung von Programmen aller Art zu Optimierungen, was zur Einsparungen von monetären und natürlichen Ressourcen führen kann und konkret bei Unternehmen den Gewinn vervielfachen kann.

Inhaltsverzeichnis

1 Einleitung

Im Informatik Leistungskurs des 12. Jahrgangs beschäftigten wir uns nach den Herbstferien mit der Laufzeit von Algorithmen am Beispiel der Quicksort, einem Sortieralgorithmus. Nach diesem thematischen Impuls ergab sich mein Projekt zur Erforschung von Sortieralgorithmen, in dem ich es mir zuerst zur Aufgabe gemacht hatte, den schnellsten Weg zu finden, 10 Millionen zufällig generierte Zahlen zu sortieren. Mein Fokus änderte sich dann allerdings und ich fokussierte mich auf die Aspekte der Implementation und tatsächlichen Ausführung der Programme, da Sortieralgorithmen algorithmisch bereits sehr weit erforscht sind und man in diesem Gebiet nur sehr schwer neue Erkenntnisse sammeln konnte [22]. Ich entschied mich deswegen, nicht nach besseren Algorithmen zu suchen, sondern nach Wegen, mein ursprüngliches Program in Python in seiner Ausführung zu beschleunigen und so vorteilhafte Wege der Geschwindigkeitsoptimierung zu entdecken. Ich stellte mir die Frage, welche Optimierungen die Ausführgeschwindigkeit von Programmen am meisten erhöhen und wieso? Zum Thema der Optimierung fand ich im Bezug auf Python einige Bibliotheken im Internet, die bessere Performance versprachen, von denen ich einige auswählte, um herauszufinden, welche dieser den größten Geschwindigkeitsboost bringt. Außerdem entschied ich mich noch dazu, einige Algorithmen auch in C++ zu implementieren, wobei ich im Bezug darauf nicht direkt Möglichkeiten und Erklärungen fand, welche Modifikationen ein Program schneller machen und wieso diese Veränderungen es schneller machen. Weiterführend verglich ich noch, wie sich meine C++ Programme mit denen in Python in der Ausführung auf einer tiefergreifenden Stufe unterschieden, um herauszufinden, wieso C++ meist deutlich schneller ist. Meine Erwartung war dabei, dass C++ bei jeder Aufgabe Python um ein vielfaches übertrifft im Punkt der Geschwindigkeit, da Python im allgemeinen als langsam gilt und C++ als sehr performant und schnell.

2 Vorgehensweise, Materialien, Methode

2.1 Vorgehen

Meine Vorgehensweise beruhte darauf, mich im Internet über Möglichkeiten der Optimierung zu informieren und diese geschickt neu zu kombinieren und die erreichten Geschwindigkeiten miteinander zu vergleichen, wodurch ich eine Qualitative Entscheidung treffen konnte, welche Variationen die größten Performanceverbesserungen bringen. Dazu wählte ich die Python Bibliotheken *numpy*, *numba*, *ctypes* und *cython* und testete auch Funktionen aus den Standardbibliotheken der Sprachen. Bei der Implementation nutzte ich Internetquellen wie Stackoverflow, die Cython Dokumentation und andere (alle Seiten aufzuzählen ist nicht zielführend), um die Implementierung umzusetzen, da ich noch kaum Erfahrung mit diesen Bibliotheken hatte [23d] [23a]. Dies wurde besonders wichtig beim implementieren in den mir noch kaum bekannten Sprachen Java, Lua, Julia und Go. Meine Implementationstechnik war das iterative Implementieren, wo ich eine Startversion immer wieder abwandelte und konstant Dinge änderte. Dabei behielt ich immer die Ausführungszeit im Auge und entschied so, welche Ansätze weitere Tests erforderten. Bei der Implementation einer Radixsort, einem sehr schnellen Sortierverfahren, nutze ich außerdem 2 Publikationen zur Verbesserung meiner eigenen Variante und schuf so die Untergrenze all meiner getesteten Implementationen [Ter00] [Her01]. Im Zuge dessen beschäftigte ich mich auch mit dem Konzept von AVX2 und nutze diesen erweiterten Befehlssatz, den ich im Rahmen der Ergebnisse noch weiter beschreiben werde. [23c] [Sca16].

2.2 Materialien

Für mein Projekt nutze ich meinen Desktop PC (Linux) und meinen Laptop (Linux) zur Implementation und Ausführung der Programme. Dabei nutze ich verschiedene Arten von Software zum entwickeln und Ausführen meiner Programme und zum erstellen meiner schriftlichen Arbeit.

Ich nutze dabei diese Software in der jeweiligen Version:

- Visual Studio Code (Editor für Code)
- LaTeX(Erstellen der Dokumentation)
- clang++ 14.0.6 (C/C++ Compiler)
- python 3.10.8 (Ausführen der Dateien)
- java 19.0.1 openjdk (Javac, JVM)
- lua 5.4.4 (Ausführen der Dateien)
- nodejs 18.8.0 (Ausführen der Dateien)
- julia 1.8.3 (Ausführen der Dateien)
- go 1.19.4 (Compilierung)

2.3 Methode

Meine Methode war das strukturierte Testen und vergleichen von Programmimplementationen und ihrer Variationen mithilfe einer einfachen Zeitname. Ich nutze die Möglichkeiten der jeweiligen Programmiersprachen, um die Ausführungszeiten auf die Milisekunde genau oder sogar genauer zu bestimmen und so zu vergleichen zu machen. In Python definierte ich mir dafür eine eigene Klasse (Abb. 1), fügte das Starten und Stoppen des Timers meinem code hinzu (Abb. 2) und ließ mir die Ergebnisse als Text ausgeben (Abb. 3). Damit die Ergebnisse vergleichbar sind, implementierte ich immer auf ähnliche Weise die Algorithmen und nutze die gleichen Eingabewerte und Konstanten. Als Beispiel generierte ich mir in jeder Sprache eine Liste aus 10 Millionen Zufallszahlen, was ich nicht in die Zeit mit einbezog, und wendete auf diese dann die implementierten Algorithmen an. So hatten alle Programme die Gleiche Aufgabe zu lösen, wodurch Vergleichbarkeit gewährleistet ist. Wichtig war auch, die finalen Tests direkt nach einem Neustart des PCs ohne andere laufende

Programme zu machen, damit es durch variierende Prozessorauslastung keine Verfälschungen gab.

```python
import time    Redstonerayy [5 weeks ago] • started restruct

class Timer:
    timers = {}
    @staticmethod
    def start(name):
        Timer.timers[name] = time.time_ns()

    @staticmethod
    def stop(name):
        diff = time.time_ns() - Timer.timers[name]
        diffseconds = diff / (10 ** 9)
        print(name, ":", round(diffseconds, 5), "Seconds")
```

Abb. 1

```python
Timer.start("standard list.sort() on numpy array")
liste.sort()
Timer.stop("standard list.sort() on numpy array")
```

Abb. 2

```
➜ numbanumpy git:(master) ✗ python standard_sort.py
standard_sort.py
standard list.sort() on numpy array : 0.88711 Seconds
standard list.sort() on python list : 5.4044 Seconds
➜ numbanumpy git:(master) ✗
```

Abb. 3

2.4 Schwierigkeiten

Die größte Schwierigkeit war, dass ich auf diesem Themengebiet noch nicht sehr viel Vorerfahrung hatte. Außerdem waren die Konzepte teils komplex und es war nicht immer einfach, im Internet gute Beispiele zur Implementierung zu finden. Vor dem Projekt hatte ich mich z. B. noch nie mit der Python Bibliothek ctypes”beschäftigt und deshalb fand ich meist erst nach längerem Suchen im Internet eine Lösung für auftretende Fehler. Auch bei der Implementation von Programmen in C++ hatte ich teils meine Schwierigkeiten, da das Verstehen von einigen Bitoperationen erstmal ein eindenken in die Thematik erforderte. Mit dem Ausführen der Programme hatte ich hingegen kaum Probleme. Ein für mich nicht lösbares Problem war auch die Implementierung in Go, wo nur dass generieren einer 7 Millionen Zahlen langen Liste möglich, da bei mehr Zahlen mehr Arbeitsspeicher benötigt wurde, als Go in einem Programmteil erlaubt. Der hohe Verbrauch an Arbeitsspeicher kommt meinen Vermutunge nach von der Implementation von Recursion in Go, die nicht für sehr Tiefe Rekursion geeignet scheint [23b] [tpa20].

3 Ergebnisse

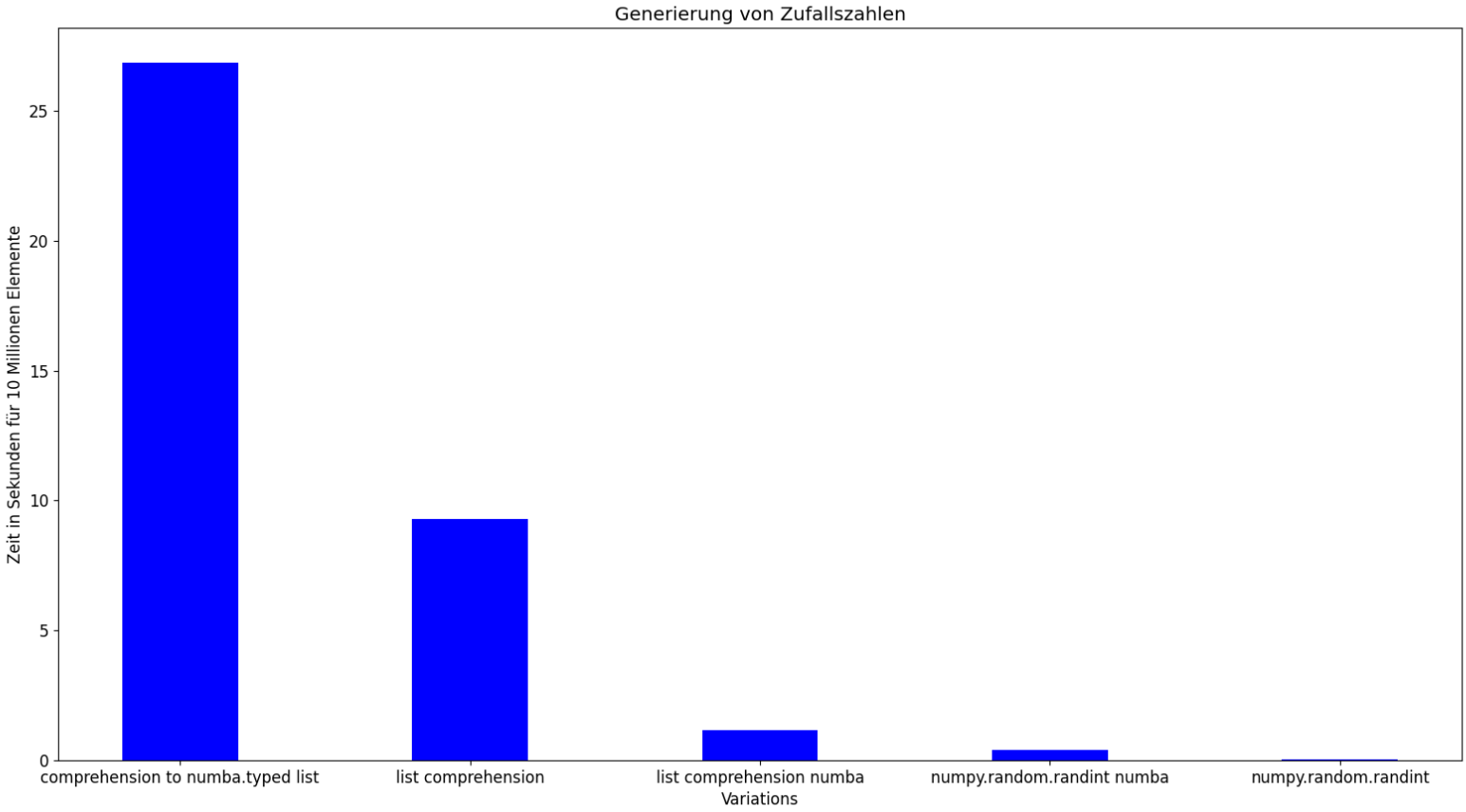

Abb. 4

Zuerst beschäftigte ich mich damit, möglichst schnell die Liste von Zufallszahlen zu generieren. Die nicht optimierten Version, die reine list comprehensionsnutzen schneiden erwartungsgemäß schlecht ab und sind deutlich langsamer als andere Varianten. Die numbanutzende Version ist ca. 5x schneller, was darauf zurückzuführen ist, dass der Ursprüngliche Python Code dem numba”JIT-Compiler übergeben wird. JIT Compilation bedeutet, dass während des Ausführens des Programms Teile des Programms wie z. B. Funktionen vom JIT-Compiler in eine optimierte Form zur Ausführung übertragen werden. Dies kostet zwar Zeit, doch bei Funktionen, die häufig ausgeführt werden oder wenn die Optimierungen eine sehr große Zeitersparnis bringen. Der 2. Aspekt ist hier der Fall, obwohl die Kompilierung Zeit kostet, wird diese bei der Ausführung deutlich eingespart. Schneller sind nur die Funktion numpy.random.randint, die ein numpy arraymit Zufallszahlen erstellt, wobei hier auffällt, dass diese durch den JIT-Compiler nicht schneller wird, was damit zusammenhängt,

dass die Kompilierung länger als die eigentliche Ausführung dauert. Die Erklärung dafür, dass diese Funktion der numpy”Bibliothek so schnell ist, ist darin zu finden, dass diese in C geschrieben wurde und bereits kompiliert wurde. Dadurch wird das numpy array”genau so schnell wie in C generiert. Diese fand ich durch betrachten des NumPy Github Repository heraus.

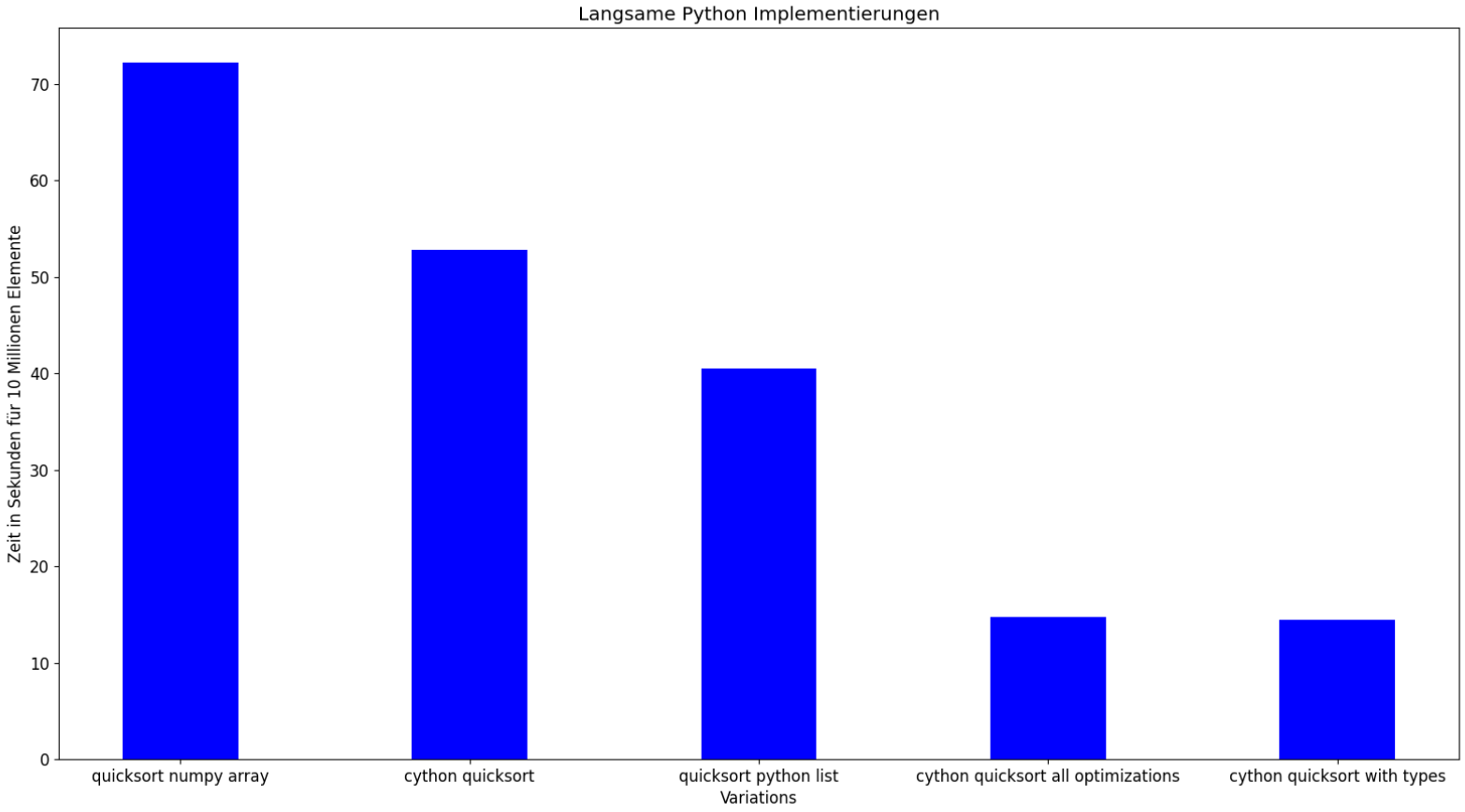

Abb. 5

Nachdem nun eine zu sortierende Liste vorhanden war, implementierte ich mehrere Versionen der Sortieralgorithmen. In Abbildung 5 sind die langsamen Versionen zu sehen, bei denen wenige Optimierungen vorgenommen wurden, oder wo diese nichts bewirkten. Ausgehend von der Implementation mit normalen Python Listen reduzierte sich die Ausführdauer von ca. 40 auf 14 Sekunden durch die Nutzung von statischer Typisierung und Kompilierung mit Cython. Das hinzufügen weiteres Optmierungen bei Cython (’all optmizations’) hatte in diesem Fall dann allerdings keine Auswirkungen mehr. Zu erklären ist die um 281% höhere Geschwindigkeit damit, dass der Cython Code vom Cython Compiler entsprechend Kompiliert werden kann und die in Cython implementierten

Funktionen somit bereits besser Optimiert sind bei ihrer Ausführung als normaler Python Code, der mühsam vom Python-Interpreter eingelesen und verarbeitet werden muss, was mehr Zeit kostet. Überraschend war für mich, dass das Nutzen von den scheinbar schnelleren 'numpy arrays' und das Kompilieren von reinem Python Code mit Cython für eine Verschlechterung sorgten, was ich mir dadurch erklärte, dass einige Python Funktionen auf normale Listen besser funktionieren als auf 'numpy arrays', da sich diese in ihrer Funktionsweise unterscheiden. Warum die Cython Version langsamer ist konnte ich mir noch nicht erklären, wobei es vielleicht an meinem noch nicht tiefgreifend genug gehenden Verständnis von Cython liegen könnte.

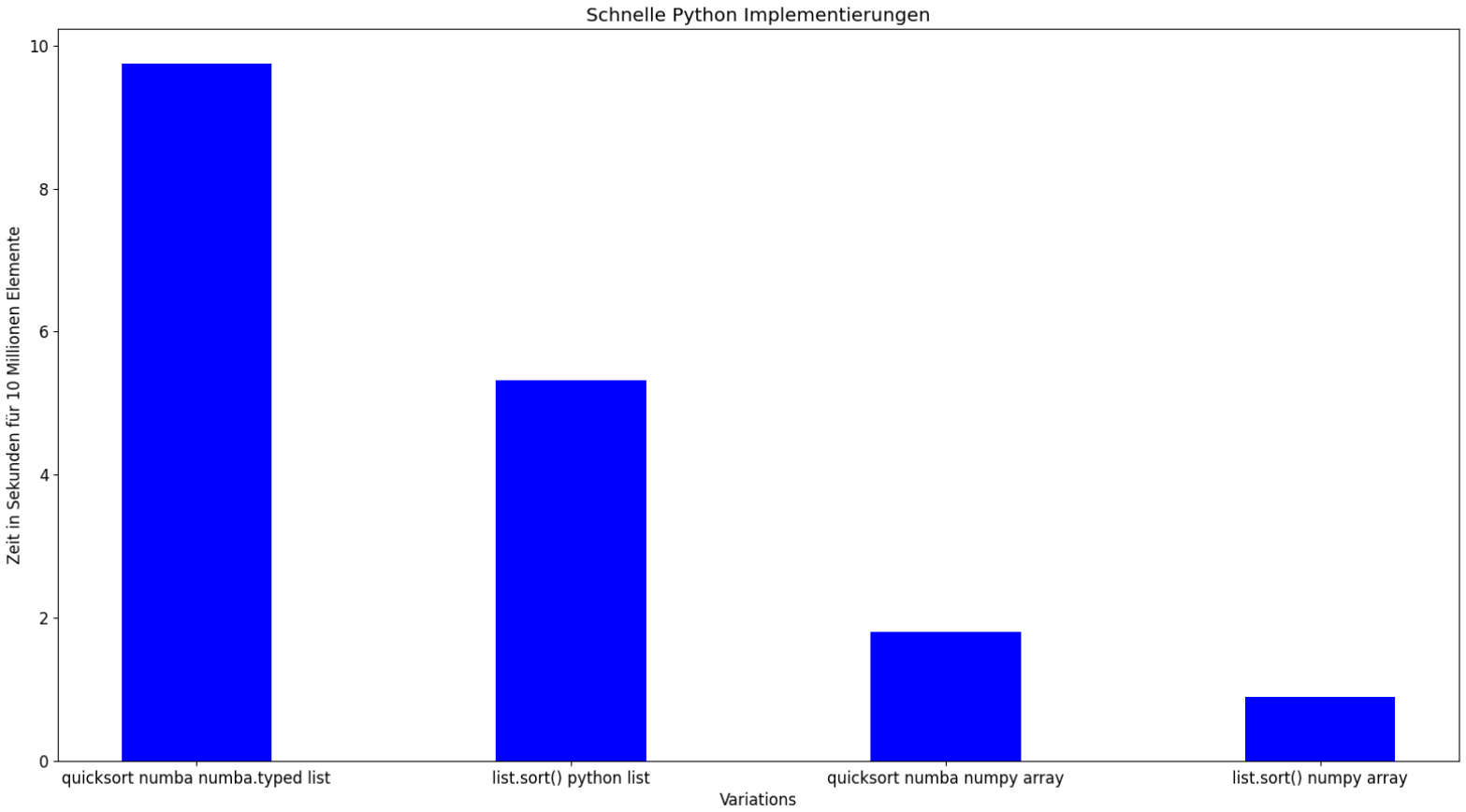

Abb. 6

Deutlich erfolgreicher und einfacher war die Implementierung mit 'numba' und 'numpy'. Es zeigte sich, dass das anwenden von JIT-Kompilierung in Form von 1er Zeile Code eine Reduzierung auf ≈ 9 und ≈ 2 Sekunden bewirkt. Diese hohe Effektivität von JIT ist aufgrund dessen, dass die Quicksort Funktion sich häufig Rekursiv aufruft, und so bei jedem Aufruf Zeit eingespart wird, was sich dann aufsummiert zu einer hohen

Zeitersparnis. Außerdem fällt auf, dass die standardmäßig in Python implementierte ’sort()’ Methode der Listen am besten abschneidet. Diese ist nämlich, wie die NumPy Funktionen und alle built-in Python Funktionen, in C geschrieben und somit sehr schnell. Mit der Kombination der in C geschriebenen Funktionen erreicht man so eine Zeit von 0.9 Sekunden.

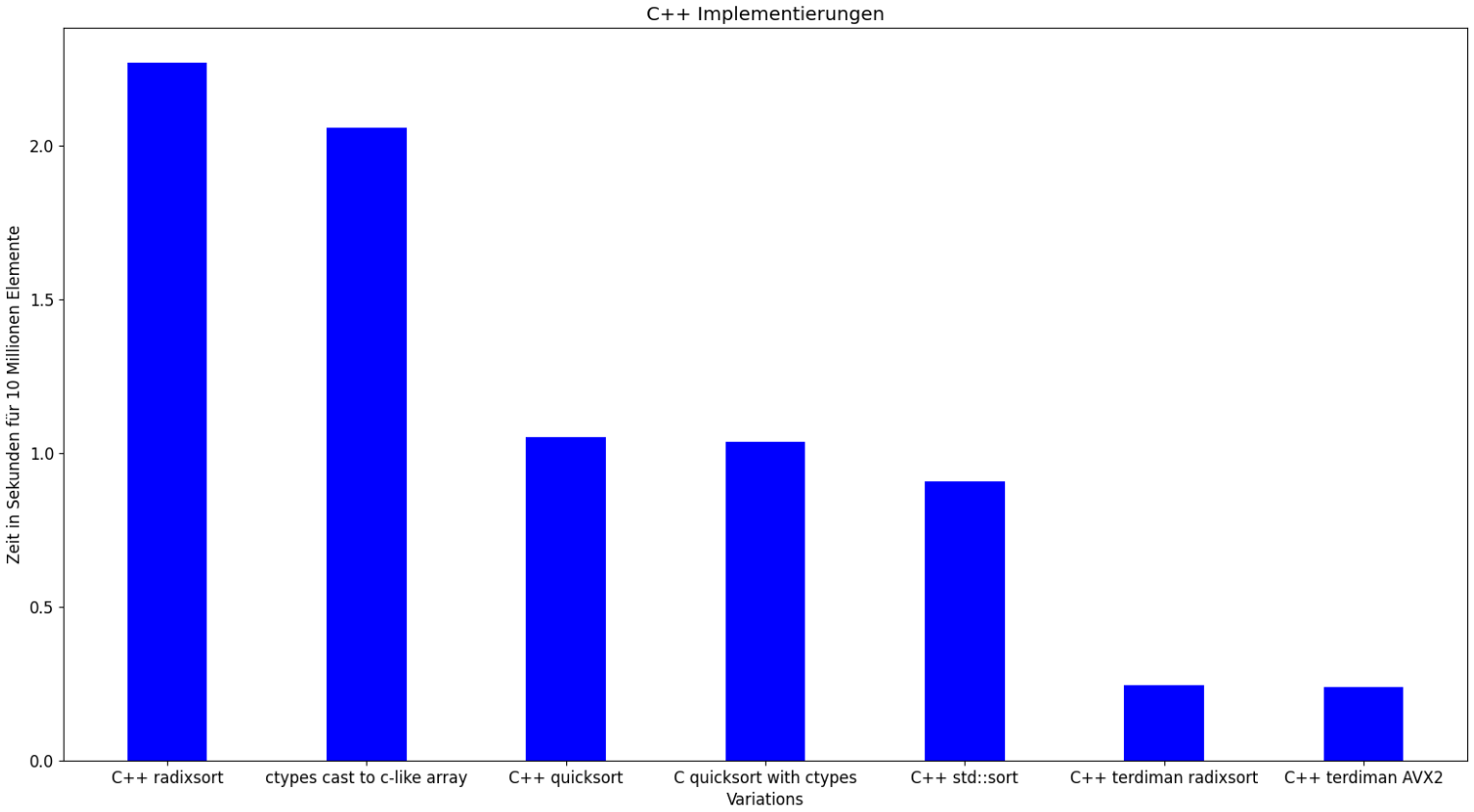

Abb. 7

Bei meiner Implementation des Quicksort Algorithmus in C++ und dem Testen der standardmäßigen Sortierfunktion viel mir auf, dass diese nahezu gleich schnell waren wie die schnellste Python implementierung. Daraus schloss ich, dass kaum ein Unterschied bestand zwischen dem, was der Computer bei meinem C++ Code und beim schnellsten Python Code machte. Die genutzten Python Funktionen waren ja schließlich auch in C geschrieben. Diese Vermutung bestätigte ich, indem ich meine in C++ Quicksort nach C umschrieb und mit dem Modul CTypes aus Python heraus aufrief und damit ähnliche Geschwindigkeiten erreichte. Es zeigt sich, dass die Geschwindigkeit von Python Code stark davon abhängt, ob Funktionen in C geschrieben wurden, die man nur noch aufrufen muss, oder ob Funktionen in Python ausgeführt werden müssen. Abschließend

gelang es mir noch, einige weitere Techniken der Optmierung in C++ zu entdecken und umzusetzen. Meine Ursprüngliche Implementation der RadixSort, einem Sortierverfahren der Komplexität $n \cdot w$, was theoretisch deutlich schneller ist als die Quicksort mit einer Komplexität von $n \cdot log_n$, stellte sich als langsamer als die Versionen der Quicksort in Python und C++ heraus.

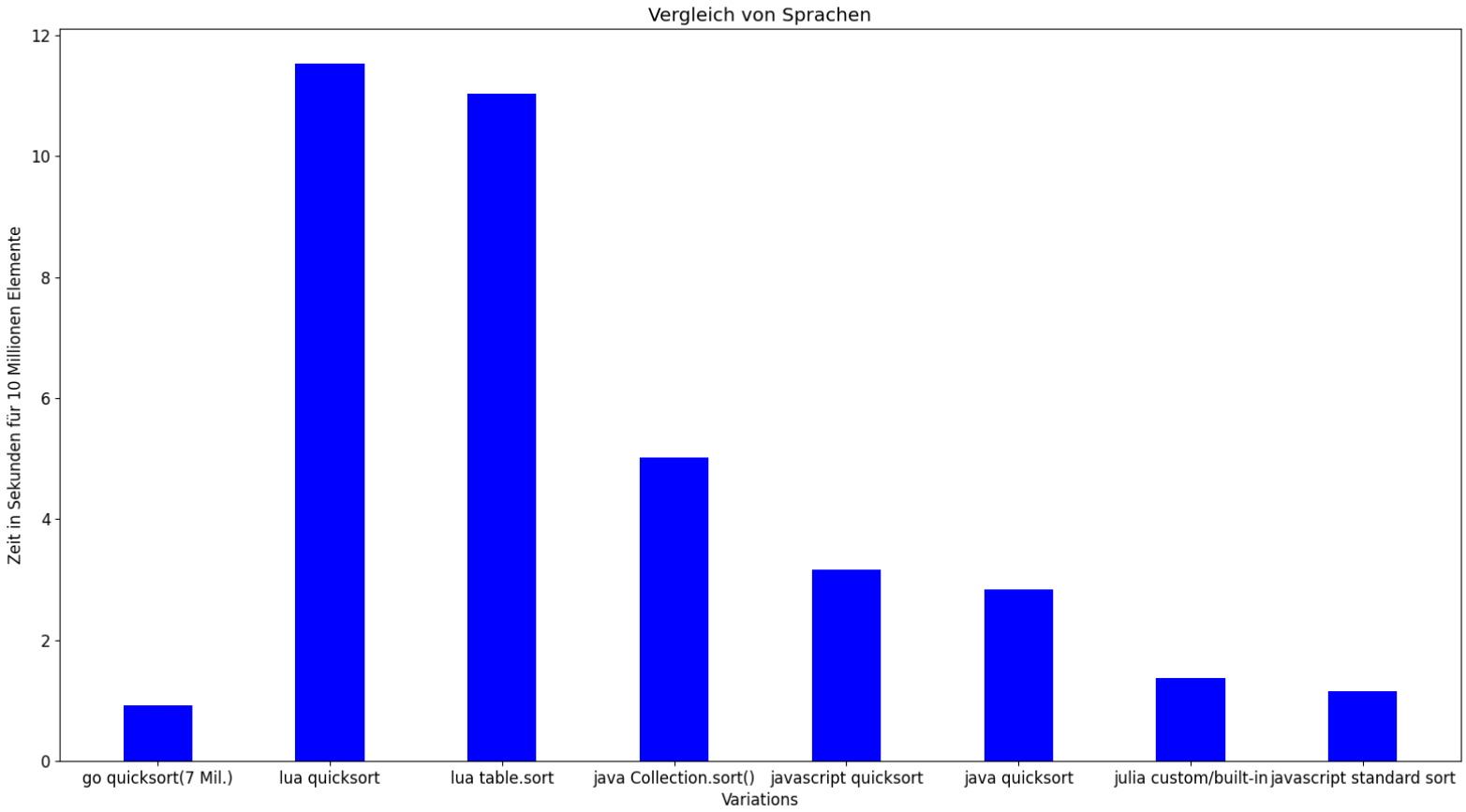

Abb. 8

Neben C++ und Python interessierte mich auch, wie schnell das Sortieren mit einer Implementation der Quicksort in anderen Sprachen möglich ist. Dazu testete ich 3 interpretierte Sprachen, nämlich Lua, Javacript und Julia sowie die Kompilierten Sprachen Go und Java, wobei Java nur in Java-ByteCode Kompiliert wird, der dann von der JVM ausgeführt wird. Dabei stellte ich fest, dass Lua im Vergleich sehr langsam ist, doch die anderen interpretierten Sprachen ähnlich schnell zu Python waren. Dies lässt sich dadurch erklären, dass die V8, also der Javascript Interpreter, im Gegensatz zum Python-Interpreter, mehr Optimierungen vornimmt, so z. B. ohne Arbeit des Entwicklers direkt JIT-Compilation. Bei Julia hingegen hatte ich diese schnelle Ausführung erwartet, da die Sprache

für wissenschaftliche Berechnungen konzipiert ist. Obwohl man Julia Programme nicht selbst kompilieren kann, so werden diese meist von der Julia Runtime vor der Ausführung kompiliert und optmiert, was zu dieser Performance führt. Die Testergebnisse der beiden Kompilierten Sprachen Java und Go sind leider nicht so aufschlussreich. Bei Go hatte ich Probleme bei der Implementierung und durch die Unmöglichkeit, in Go eine echte, rekursive Quicksort mit 10 Millionen Zahlen zu implementieren ist dass Ergebnis nur Schätzbar. Ich würde die Geschwindigkeit ähnlich der wie Julia einschätzen, da 3 Millionen Zahlen weniger sortiert wurden. Bei Java scheint sich auf den ersten Blick einer eher mäßige Performance abzuzeichnen, doch sollte man im Hinterkopf behalten, dass ich sehr wenig Erfahrung mit Java habe, weshalb eine Implementation in Java möglicherweise schneller sein könnte. Insgesamt zeigt sich, dass die Interpreter für Python und Lua nicht so gut optimiert sind, wobei man beachten muss, dass Projekte wie NumPy und Numba sowie LuaJIT, die Geschwindigkeit dieser deutlich erhöhen können.

4 Diskussion

5 Zusammenfassung

Literaturverzeichnis

[Ter00] Pierre Terdiman. „Radix Sort Revisited“. In: (2000). URL: http://codercorner.com/RadixSortRevisited.htm (besucht am 11. 01. 2023).

[Her01] Michael Herf. „Radix Tricks“. In: (2001). URL: http://stereopsis.com/radix.html (besucht am 11. 01. 2023).

[Sca16] Matt Scarpino. „Crunching Numbers with AVX and AVX2“. In: (2016). URL: https://www.codeproject.com/articles/874396/crunching-numbers-with-avx-and-avx.

[tpa20] tpaschalis. „What is a goroutine? What is their size?“ In: (2020). URL: https://tpaschalis.me/goroutines-size/.

[22] *Sortierverfahren*. 2022. URL: https://de.wikipedia.org/wiki/Sortierverfahren (besucht am 11. 01. 2023).

[23a] *Cython Dokumentation*. 2023. URL: http://docs.cython.org/en/latest/.

[23b] *Go stack limit reached on deep Recursion*. 2023. URL: https://stackoverflow.com/questions/69625277/runtime-goroutine-stack-exceeds-1000000000-byte-limit.

[23c] *Intel AVX2 Dokumentation*. 2023. URL: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/intrinsics/intrinsics-for-avx2/intrinsics-for-shuffle-operations-1/mm256-shuffle-epi8.html.

[23d] *Stackoverflow*. 2023. URL: https://stackoverflow.com/.