LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 189 – Abgabe zu Aufgabe A319
Sommersemester 2022

Lars Christiansen Leon Kogler Leopold Bauer

1 Einleitung

1.1 Komplexe binäre Zahlensysteme

Das Rechnen mit komplexen Zahlen spielt heut zu Tage in vielen Anwendungen, wie zum Beispiel der Daten und Bildverarbeitung, eine wichtige Rolle[2, S. 1]. In solchen Gebieten sind schnelle arithmetische Operationen auf komplexen Zahlen von großer Bedeutung. Ohne einem komplexen binären Zahlensystem müssten Real- und Imaginärteil jeweils separat berechnet werden um zum Beispiel eine Addition oder eine Subtraktion durchzuführen(1). Weiters entstünde bei der Multiplikation bzw. Division komplexer Zahlen ein erheblicher Mehraufwand, da Beispielsweise bei der Multiplikation jeweils jeder Real- und Imaginärteil der ersten Zahl mit jedem Real- bzw. Imaginärteil der Zweiten Zahl multipliziert wird und anschließen auf jedem Produkt der Multiplikationen eine Addition ausgeführt werden muss(2). Insbesondere die Division stellt einen großen Aufwand zur Berechnung dar.

$$(a + ib) + (c + id) = (a + c) + i(b + d) \tag{1}$$

$$(a + ib) * (c + id) = ac + i(ad + bc) + i^2bd = (ac - bd) + i(ad + bc) \tag{2}$$

Verwendet man statt der üblichen Darstellung komplexer Zahlen ein binäres Zahlensystem, welches es ermöglicht Real- und Imaginärteil in einer einzigen binären Zahl abzuspeichern und arithmetische Operationen in einer Rechenoperation durchzuführen, so könnte dies das Operieren mit komplexen Zahlen auf Computern erheblich beschleunigen. Außerdem könnten so Mikroprozessoren entwickelt werden, welche auf arithmetische Operationen mit komplexen binären Zahlen optimiert sind. Theoretisch könnte so, im Vergleich zum normalen rechnen mit komplexen Zahlen, eine Beschleunigung von 50% - 800% erzielt werden. [2, S. 2-3]

1.2 Complex Binary Number System (CBNS)

Auf der Suche nach einer geeigneten Basis um komplexe Zahlen effizient in binärer Form darzustellen, präsentierte Tariq Jamil et al. im Jahr 2000 die Basis $-1+i$ und Algorithmen, mit welcher arithmetische Operationen, insbesondere die Division, effizient berechnet wurden konnten[1, S. 268–274]. Dieses komplexe binäre Zahlensystem mit Basis $-1 + i$ ist auch unter dem Namen „Complex Binary Number System“ (CBNS) bekannt. [2, S. 3-4]
Folgend werden die ersten 8 Basen von $-1 + i$ aufgelistet und eine Abbildung, welche

demonstriert wie dieses Zahlensystem die Zahlenebene der Komplexen Zahlen abdeckt, abgebildet.

$$(-1+i)^0 = 1$$
$$(-1+i)^1 = -1+i$$
$$(-1+i)^2 = -2i$$
$$(-1+i)^3 = 2+2i$$
$$(-1+i)^4 = -4$$
$$(-1+i)^5 = +4-4i$$
$$(-1+i)^6 = 8i$$
$$(-1+i)^7 = -8-8i$$

Abbildung 1 zeigt wie sich die komplexe Zahlenebene mit der Basis $-1+i$ abdecken lässt. Betrachtet man die Basis hoch 0, also das erste binäre Bit in einem CBNS, so kann man in der Abbildung 1 in rot markiert die beiden Punkte $0+0i$ für binär 0_{-1+i} und $1+0i$ für binär 1_{-1+i} erreichen. Weiter's gilt dies auch für die Basis hoch 1 in dunkelgrün, für die Basis hoch 2 in orange, für die Basis hoch 3 in gelb usw.. Durch die wiederholte Multiplikation der Basis $-1+i$ mit sich selbst ergibt sich so zwischen jeder Basis $(-1+i)^n$ und $(-1+i)^n+1$ eine Winkelrotation um 135° im Uhrzeigersinn und eine Erhöhung des Betrags der Länge zum Nullpunkt um den Faktor $\sqrt{2}$. Wie in Abbildung 1 zu sehen ist, entsteht so ein sich entwickelndes Fraktal, welches die komplexe Zahlenebene zwar lückenlos abdecken kann, sich jedoch nicht gleichmäßig ausbreitet. Somit kommt es zu den Fällen, dass sich Beispielsweise die komplexe Zahl $3+2i$ mit einer 4 Bit CBNS Zahl ohne Probleme als

$$1001_{-1+i} = 1*(-1+1)^3 + 0*(-1+1)^2 + 0*(-1+1)^1 + 1*(-1+1)^0$$

darstellen lässt, sich jedoch die Zahl -1_{-1+i} im CBNS erst ab mindestens fünf Bits darstellen lässt.

$$11101_{-1+i} = 1*(-1+1)^4 + 1*(-1+1)^3 + 1*(-1+1)^2 + 0*(-1+1)^1 + 1*(-1+1)^0$$

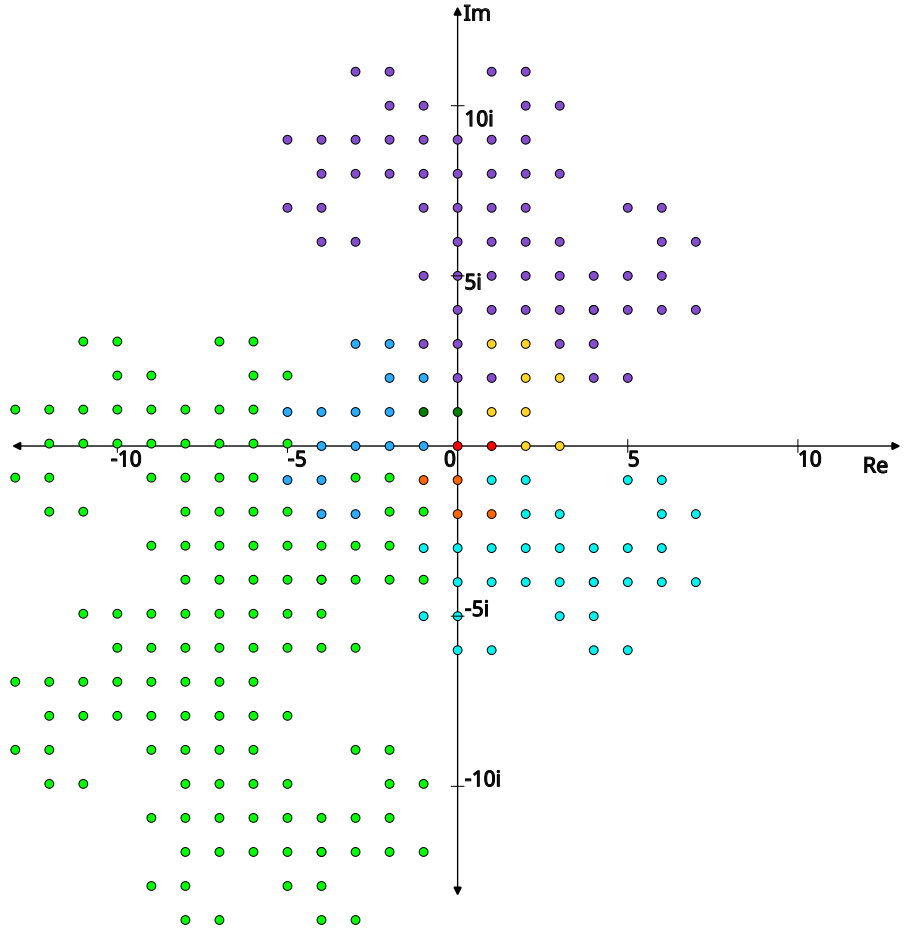

Abbildung 1: Abdeckung der komplexen Zahlenebene mit den ersten 8 Basen von $-1+i$ $(-1+i)^0$ in Dunkelrot, $(-1+i)^1$ in Dunkelgrün, $(-1+i)^2$ in Orange, $(-1+i)^3$ in Gelb, $(-1+i)^4$ in Blau, $(-1+i)^5$ in Türkis, $(-1+i)^6$ in Violett, $(-1+i)^7$ in Hellgrün

2 Lösungsansatz

2.1 Umwandlung binärer Zahlen zur Basis -1+i in komplexe Zahlen

Die Umwandlung binärer Zahlen zur Basis -1+i in komplexe Ganzzahlen funktioniert analog zur Umwandlung von binären Zahlen zur Basis 2 ins Dezimalsystem. Wie auch für andere Zahlensysteme, eignet sich die Formel 2.1 zur Umwandlung von binären Zahlen zur Basis $-1+i$ in Dezimalzahlen zur Basis 10.

$$n = \sum_{j=0}^{N-1} b_j * (B)^j$$

- n... die Zahl nach der Umwandlung

- N... die Anzahl der Stellenwerte der zu umwandelnden Zahl

- b_j... der Wert an der Stelle j

Solange die jeweiligen Basen der jeweiligen Stellenwerte gegeben sind entspricht die Umwandlung in eine komplexe Zahl nur der Aufsummierung der einzelnen Real- bzw. Imaginärteile der jeweiligen Stellenwerte.

2.1.1 Algorithmische Umsetzung

Um eine binäre Zahl im CBNS in eine komplexe Zahl umzuwandeln wurde die Funktion $to\_carthesian$ eingeführt, welche einen $unsigned\_\_int128$ in seine Real- und Imaginärteile aufteilt und an zwei $\_\_int128$ Pointer dem Speicher übergibt. Dabei wird die oben genannte Formel 2.1 auf jeweils den Real- bzw. Imaginärteil angewandt. Um die jeweiligen Stellenwerte der binären Zahl zu bestimmen wurden zunächst die ersten acht Stellenwerte des Real- und Imaginärteils in zwei Array abgespeichert. Dabei wird ausgenützt, dass sich der Stellenwert an der Stelle j alle acht Stellen versechszehnfacht. Wie in Kapitel 1.2 bereits festgestellt wurde entsteht je Multiplikation des vorherigen Stellenwertes mit der Basis $-1 + i$ eine Winkelrotation um 135° und eine Betragsänderung der Länge zum Nullpunkt um den Faktor $\sqrt{2}$. Somit ergibt sich nach acht Multiplikationen eine Rotation von $8 * 135° \mod 360 = 0°$ und eine Betragsänderung der Länge um $8 * \sqrt{2} = 2 * 2 * 2 * 2 = 16$. Um nun auch die Werte nach den ersten acht binären Stellen zu bestimmen werden die Einträge der beiden Arrays, nach allen 8 Stellen in der binären Zahl, mit 16 Multipliziert. Nun können alle Stellenwerte von b_0 bis b_{N-1} mithilfe einer einfachen Schleife aufaddiert werden und am Ende des Algorithmus den Pointern übergeben werden.

2.1.2 Algorithmische Umsetzung der Vergleichsimplemtierung

Als Vergleichsfunktion wurde $to\_carthesian\_V2$ erstellt, welche die Formel 2.1 strikt befolgt und dabei für jedes gesetzte Bit die Methode add_bin_at aufruft. Diese Version verwendet jedoch einen einfacheren Algorithmus, welcher allerdings auch ungenauer und langsamer ist. So können hier nur CBNS Zahlen bis zu 32 Bit konvertiert werden. Dies kommt daher, dass hierbei das Pascalsche Dreieck verwendet wird und dabei die Zahlen sehr groß werden können. Der Algorithmus geht folgendermaßen vor: Zuerst wird mit $pascals\_triangle$ das Pascalsche Dreieck berechnet, welches daraufhin mit $adjust\_minus$, um die Subtraktion in der Basis zu beachten, manipuliert wird. Nun werden noch die Komplexen Zahlen mit $shorten\_i$ gekürzt und anschließend der Imaginär- und Realteil separat in $count\_together$ zusammengezählt.

2.2 Umwandlung komplexer Zahlen in eine binäre Zahl zur Basis -1+i

Die Umwandlung in die entgegengesetzte Richtung ist ein wenig komplizierter und damit auch Zeitaufwendiger. Damit man die komplexe Zahl in binärer Form darstellen kann muss man den komplexen und realen Teil getrennt betrachten und jeweils vier Schritte durchführen, bis man danach die Zahl wieder zusammenführen kann.
Für ein besseres Verständnis wird als Beispiel die Zahl $60 + 2000i$ konvertiert.
Anfangs betrachten wir somit die Zahlen 60 und 2000 und vermerken, dass 2000 der imaginäre Teil der Zahl war. Ist die Zahl negativ, so wird dies notiert und der absolute Wert genommen.

(1) Als allererstes wird die Zahl in die äquivalente Zahl zur Basis 4 umgewandelt. Beispiel:

$$60 = (3, 3, 0)_4$$
$$2000 = (1, 3, 3, 1, 0, 0)_4$$

(2) Nun wird die Zahl weiter zur Basis -4 umgerechnet. Dies wird aufgrund dessen, dass wir die Zahl bereits in der Basis 4 vorhanden haben, sehr viel vereinfacht, da nur jede Ziffer an einer ungeraden Position mit -1 multipliziert werden muss. Allgemein mit n gerade:

$$(z_n, z_{n-1}, ..., z_3, z_2, z_1, z_0)_4 = (z_n, -z_{n-1}, ..., -z_3, z_2, -z_1, z_0)_{-4}$$

Beispiel:

$$(3, 3, 0)_4 = (3, -3, 0)_{-4}$$
$$(1, 3, 3, 1, 0, 0)_4 = (-1, 3, -3, 1, -0, 0)_{-4}$$

(3) Der dritte Schritt besteht darin die Zahl zu normalisieren. Dabei ist das Ziel, dass alle Zahlen in dem Tupel danach einen Wert zwischen eins und drei besitzen. Um dies zu erreichen wird zu jeder negativen Zahl an der Stelle n $+4$ und gleichzeitig $+1$ zu der Zahl $n + 1$ links von der ursprünglichen Zahl addiert. Dabei können jedoch Zahlen entstehen, welche größer als drei sind. Um auch diese wiederum zu normalisieren wird diese Zahl mit der Position n auf 0 gesetzt und die Zahl $n + 1$ auf der linken, identisch dem anderen Fall, verändert. Jedoch wird hier die 1 nicht addiert sondern subtrahiert. Dabei kann es natürlich dazu kommen, dass die Zahl auf der Linken ins negative rutscht. In diesem Fall wiederholt man die eben genannten Schritte so lange, bis alle Zahlen normalisiert sind. Dies kann zu einer Schleife führen, welche jedoch terminiert.

Beispiel:

$$(3, -3, 0)_{-4} = (4, 1, 0) = (-1, 0, 1, 0) = (1, 3, 0, 1, 0)_{Normalisiert}$$
$$(-1, 3, -3, 1, -0, 0)_{-4} = (1, 3, 4, 1, 1, 0, 0) = (1, 2, 0, 1, 1, 0, 0)_{Normalisiert}$$

Basis -4	CBNS Repräsentation
0	0000
1	0001
2	1100
3	1101

Tabelle 1: Äquivalente Darstellung einer Normalisierten Zahl zur Basis -4 und der dazugehörigen Binären in CBNS [2]

(4) Zuletzt wird jede Ziffer in die dazu äquivalente binäre Darstellung in CBNS nach Tabelle 1 umgewandelt und aneinandergereiht.

Beispiel:

$$(1,3,0,1,0)_{Normalisiert} = 0b000111000000000010001000000000_{(-1+i)}$$
$$(1,2,0,1,1,0,0)_{Normalisiert} = 0b00011101000000010000_{(-1+i)}$$

Zuletzt werden die beiden konvertierten Zahlen noch zusammengeführt. Bevor dies allerdings geschehen kann muss man zuerst den imaginären Teil, im Falle, dass er positiv war mit $0b11$ oder andernfalls mit $0b111$ multiplizieren. Der reelle Anteil wird nur unter der Bedingung, dass er negativ war mit $0b11101$ multipliziert. Nun muss man die beiden CBNS Zahlen noch addieren und bekommt so die konvertierte Komplexe Ganzzahl.

2.2.1 Algorithmische Umsetzung

Bei der Hauptimplementierung sezt man nun genau dies um. Am Anfang berechnet is_abs den absoluten Wert der Zahl und speichert, ob die Zahl negativ war. Daraufhin wird für jeden Teil der Zahl getrennt in $to\_bm1pi\_positive$ ein Array, welches unser Tupel darstellt, mit der zurückgegebenen Länge von $find\_array\_len$ erstellt und der Wert des Teils zur Basis 4, welcher mit $tob4$ herausgefunden wird, hineingeschrieben. $tobm4$ wandelt es daraufhin zur Basis -4 um und $normalize$ normalisiert die Zahl solange, bis $is\_normalized$ wahr zurück gibt. Zum Schluss werden die Zahlen in ihrer binären CBNS Darstellung mit $combine\_array\_nums\_to\_bin$ noch aneinandergereiht und in to_bm1pi je nachdem ob die Zahl negativ bzw der Imaginärteil war mit den entsprechenden Faktoren multipliziert. Die zusammenaddierte Zahl wird zurückgegeben.

2.2.2 Algorithmische Umsetzung der Vergleichsimplemtierung

Die Umsetzung der Vergleichsimplementierung geht nach der Bruteforce Methode vor. Es wird eine Zahl genommen. Mit $to_carthesian$ wird die Zahl in eine komplexe Zahl umgerechnet und mit der übergebenen Zielzahl verglichen. Sollten diese gleich sein, so wird die Zahl zurückgegeben. Bei Ungleichheit wird die Zahl um eins erhöht und der Vorgang von vorne begonnen.

2.3 Komplexe binäre Addition und Multiplikation

2.3.1 Addition

Für die Umrechnung in die binäre Repräsentation von komplexen Ganzzahlen ist es wie oben schon erwähnt erforderlich, dass die binäre Addition und Multiplikation möglich und effizient durchführbar sind. Um in die Thematik einzusteigen ein kleines Beispiel: Angenommen man will folgende komplexe Berechnung in binär durchführen:

$$(-1+i)+(0-2i)=(-1+i) \tag{3}$$

Also:

$$\begin{array}{r} 0000\ 0010 \\ +\ 0000\ 0100 \\ \hline 0000\ 0110 \end{array}$$

In dem Fall konnten wir wie gewohnt mit der üblichen binären Addition die Bits bitweise aufaddieren. Der Leser kann sich nun selbst mit der Gegenprobe überzeugen, dass $(-1+i)=0010$ und $(0-2i)=0100$ entspricht und das Ergebnis 0110 sein muss. Interessanter wird das ganze bei der nächsten Addition wenn man

$$(-1-i)+(0-2i)=(-1-3i) \tag{4}$$

berechnet. Hier müssen wir nun auch wie bei der Addition von realen Binärzahlen einen Carry berücksichtigen, welcher aber nicht nur ein Bit umfasst, sondern wegen der besonderen Basis $(-1-i)$ länger ist. Wenn wir die ganze Rechnung von der anderen Seite betrachten und $-1-3i$ in binär darstellen $= 110010$ so können wir erkennen, dass wir einen Carry von 110 benötigen. Daraus ergibt sich nun folgende Wahrheitstabelle für die Addition von komplexen Binären Zahlen:

Bit 1	Bit 2	Carry	Resultat
0	0	0b000	0
1	0	0b000	1
0	1	0b000	1
1	1	0b110	0

Tabelle 2: Wahrheitstabelle zur Addition [2]

Es gibt aber noch einen Fall, der genauer Betrachtet werde muss. Und zwar ergibt sich nun ein Problem wenn man zum Beispiel 0011 mit 0111 addieren will. Hier würde eine endlose Schleife durch die carry bits entstehen, obwohl das Ergebnis von $(0+i)+(0-i)=0$ wohl definiert ist. Verallgemeinert lässt sich also folgende Gleichung zeigen:

$$(-1+i)^x+(-1+i)^{(x+1)}=-1*((-1+i)^x+(-1+i)^{(x+1)}+(-1+i)^{(x+2)}) \tag{5}$$

Daher führen wir noch eine Zusätzliche Regel ein, die sogenannte "Zero Rule"[2]. Das heißt immer wenn eine Bitfolge der Art $0b011$ und $0b111$ addiert werden, soll das Ergebnis $0b000$ sein.

Nun ein Beispiel um das ganze zu erläutern, es sollen zwei binär Komplex Zahlen addiert werden:

$$00010111 = (1 + 0i) + (-1 + i) + (0 - 2i) + (-4 + 0i) = (-4 - i) \tag{6}$$

$$00111110 = (-1 + i) + (0 - 2i) + (2 + 2i) + (0 - 4i) + (4 - 4i) = (1 - 3i) \tag{7}$$

Das Ergebnis mit Zero Rule und Carry angewandt ist also wie folgt:

$$\begin{array}{r} 0001\ 0111 \\ +\ 0011\ 1110 \\ \hline 1110\ 0001 \end{array}$$

Das Ergebnis lässt sich nun wieder leicht mit der Gegenprobe überprüfen da $11100001 = -3 - 4i$ gilt.

2.3.2 Multiplikation

Nachdem nun auf die Addition eingegangen wurde soll anschließend nun noch kurz die Multiplikation erörtert werden. Für diese haben wir die Grundlagen schon gelegt, da die Multiplikation als eine Abfolge von Additionen aufgefasst werden kann. Daher können wir auch genau gleich zu der Multiplikation mit realen Ganzzahlen umgehen: Dafür nehmen wir jedes Bit der ersten Zahl und verrunden die zweite Zahl mit diesem und schieben die Bits dieser Zahl anschließend um die Stelle des aktuellen Bits nach links. Abschließend addieren wir all diese Ergebnisse zu einer Zahl und bekommen so das Ergebnis der Multiplikation Das ganze lässt sich wieder am leichtesten mit einem Beispiel erklären:

$$\begin{array}{r} 0000\ 0101 \\ *\ 0000\ 0011 \\ \hline 0000\ 0011 \\ +\ 0000\ 0000 \\ +\ 0000\ 1100 \\ \hline\hline 0000\ 1111 \end{array}$$

Um das Ergebnis zu überprüfen rechnen wir: $(1 - 2i) * (i) = 2 + i = 1 + (-1 + i) + (-2i) + (2 + 2i)$.

2.3.3 Algorithmische Umsetzung

Aus den vorgestellten Methoden kann nun leicht ein Algorithmus für die Addition und Multiplikation abgeleitet werden. Für die Addition gehen wir beide Zahlen Bit für Bit durch, überprüfen auf die Zero Rule und aktualisieren beide Zahlen nach Bedarf. Anschließend wird in der Wahrheitstabelle abgefragt welcher Wert und welches Carry gerade zu den beiden Bits gehören und das Ergebnis daran angepasst. Wichtig ist dabei zu beachten, dass um das Carry zu addieren die Methode rekursiv aufgerufen wird, da bei der Addition des Carrys wiederum neue Carry operation entstehen können. Ein Stack

Overflow ist jedoch nicht zu befürchten, da die maximale Rekursionstiefe 128 betragen kann, die die Zahlen mit denen wir Rechnen nicht länger sein können. Hier kann leider nicht weiter durch SIMD optimiert werden, da die jeder Schritt aufeinander Aufbaut und nicht parallelisiert werden kann. Eine Möglichkeit der Optimierung besteht dennoch darin, ohne Rekursion auszukommen und den Carry in einem Array zu speichern. Dies würde den Verwaltungsaufwand auf dem Stack nochmals deutlich reduzieren, aber dazu führen, dass z.B die Zero Rule etwas schwieriger anzuwenden wäre.

3 Korrektheit

Die Korrektheit der Implementierungen der Umrechnungsfunktionen wurde zum einen über den Vergleich der Zweitimplementierungen geprüft, als auch mit einer Vielzahl an selbst geschriebenen Tests. Auch wurde dabei Ausgenützt, dass die beiden Funktionen, Umkehrfunktionen von sich selbst sind. Somit konnte mit zufälligen Eingabewerten überprüft werden, ob sich, nach Anwendung der $to\_carthesian$ Funktion und der Anschließenden Ausführung der to_bm1pi Funktion, der selbe Anfangswert ergibt. Zusätzlich wurden vor allem Randfälle überprüft, wie zum Beispiel die maximalen Eingabewerte der Funktionen. Es wurde bei der Implementierung natürlich darauf geachtet, dass zu große Eingaben in das Programm nicht zu *undefined behavior* führen, und wurden deshalb vor der Umwandlung abgefangen.
Die maximale Eingabe der Hauptimplementierung an die $to\_carthesian$ Funktion ist auf $2^{128} - 1$ begrenzt, was auch dem maximalen Übergabewert des $unsigned\_\_int128$ an die Funktion entspricht. Die Zweitimplementierung der $to\_carthesian$ Funktion ist jedoch auf $2^{32} - 1$ beschränkt.
Die maximale Eingabe an die to_bm1pi Funktion konnte nicht klar definiert werden, da wie in Abbildung 1 zu sehen ist, sich die Abdeckung, der mit 128 Bit darstellbaren Zahlen über die Komplexe Zahlenebene, nicht gleichmäßig ausbreitet. Deshalb wurde eine maximale Eingabegrenze eingeführt, welche zwar sicherstellt, dass alle Real- bzw. Imaginärwerte in eine 128 Bit große CBNS Zahl passen, jedoch nicht, dass die optimale maximale Eingabegröße erreicht wird. Somit wurde auf die letzten acht Stellenwerte der maximal darstellbaren CBNS Zahl verzichtet. Als maximaler Input wurde somit der Real- bzw. Imaginärteil der CBNS Zahl 2^{120} herangezogen, was den Werten $\pm 2^{60}$ im Real- und $\pm 2^{60}$ im Imaginärteil entspricht.

4 Performanzanalyse

4.1 Performanzanalyse Umwandlung CBNS zu Komplex

In Abbildung 2 ist die Laufzeit zum Umwandeln von N Bits einer CBNS Zahl in eine komplexe Dezimalzahl in Millisekunden dargestellt. Dabei wurde jeweils im Abstand von fünf Stellenwerten mit der maximal darstellbaren binären Zahl (Somit erste Messung mit 1, zweite mit 1.1111, dritte mit 11.1111.1111...) Messungen aufgestellt. Dabei wurde der zeitliche Mittelwert von 1000 Funktionsaufrufen je Messung aufgezeichnet.

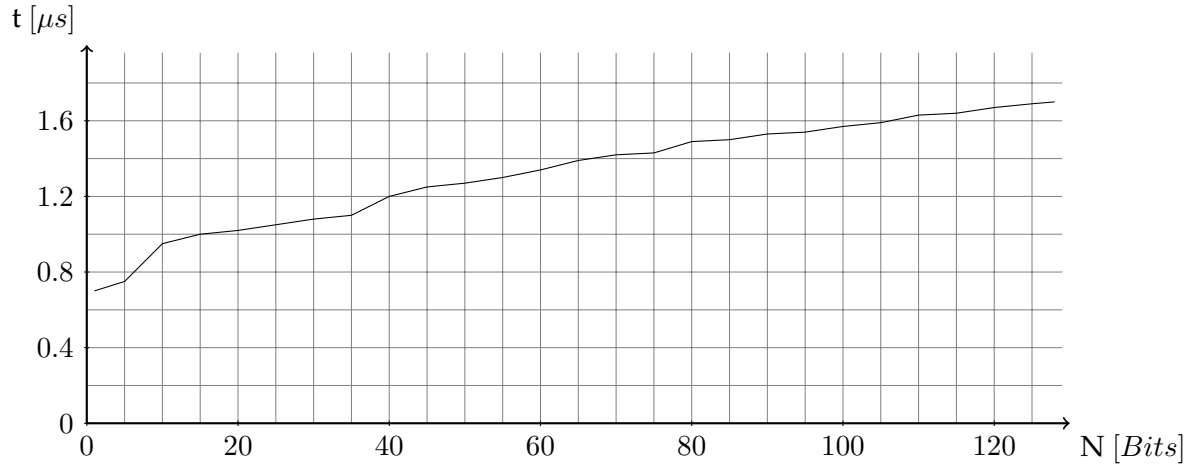

Abbildung 2: Performanzeanalyse der $to\_carthesian$ Funktion

Aus Abbildung 2 lässt sich eine lineare Laufzeit zur Umwandlung von N Bit CBNS Zahlen erkennen. Für die maximale Anzahl an Bits, also ein maximaler $unsigned\_\_int128$ benötigt der Algorithmus um die $1,7\mu s$.

4.2 Performanzanalyse Umwandlung Komplex zu CNBS

In Abbildung 3 ist die Performance der Hauptimplementierung der to_m1pi Funktion dargestellt. Dabei wurde der zeitliche Mittelwert von 1000 Funktionsaufrufen je Messung aufgezeichnet. N stellt dabei die Größe der binären Eingabe dar. Es wurde immer mit positiven Eingaben gearbeitet, und Real- und Imaginärteil waren jeweils identisch.

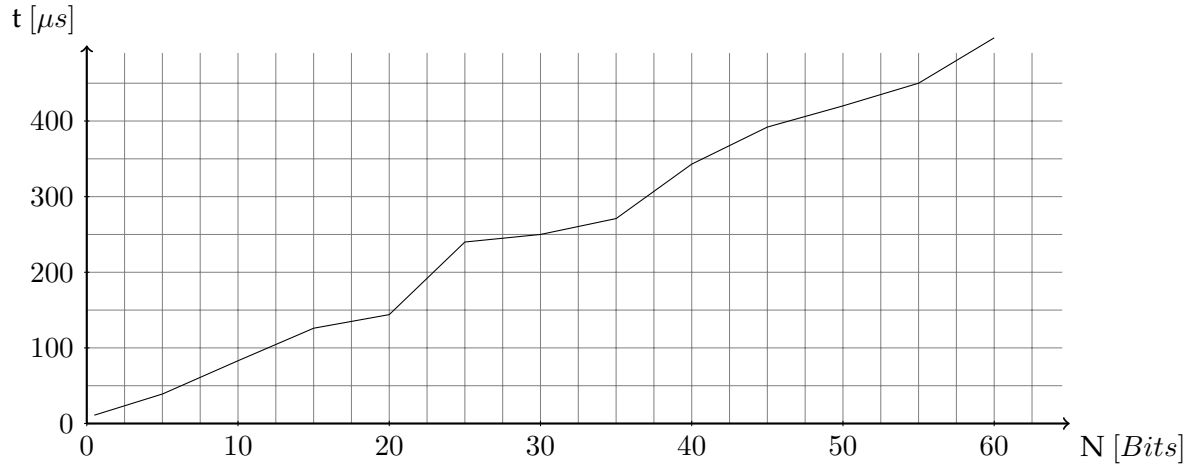

Abbildung 3: Performanzeanalyse der to_bm1pi Funktion

5 Zusammenfassung und Ausblick

Komplexe binäre Zahlen sind mit ein wichtiger Teil heutiger effizienter Algorithmen und Rechnerarchitekturen und finden Anwendung in zahlreichen Themengebieten der Informatik. Erst in den 2000er Jahren wurde mit dem CBNS ein effizentes Zahlensystem zur Berechnung komplexer binärer Zahlen gefunden[2, S. 3-4]. Die Umsetzung eines in der Programmiersprache C entwickelten Algorithmus zur Umrechnung komplexer Zahlen in eine Zahl im CBNS erforderte detailliertes Wissen wie das komplexe Binärsystem mit Basis $-1 + i$ aufgebaut ist und wie arithmetische Operationen darauf ausgeführt werden können. Dabei stellt sich die Umwandlung von komplexen Zahlen ins CBNS als wesendlich komplexer dar als die Umkehroperation. Auch Laufzeittechnisch liegen Größenordnungen zwischen den Beiden Algorithmen zur Umwandlung.

Wie Tariq Jamil in seinem Buch zeigt, könnten die Umwandlungsalgorithen auch auf Brüche oder Gleitkommazahlen angewandt werden[2, S. 8-11]. Dies wäre eine mögliche Erweiterung der aktuellen Algorithmen um genauere komplexe Werte umzuwandeln.

Literatur

[1] T. Jamil and N. Holmes and D. Blest. *Towards implementation of a binary number system for complex numbers*. IEEE, 2000. `https://ieeexplore.ieee.org/document/845574`, visited 2022-07-17.

[2] Tariq Jamil. *Complex Binary Number System*. Springer, Muscat, 2013. `https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/Complex%20Binary%20Number%20System_%20Algorithms%20and%20Circuits%20%5BJamil%202012-10-04%5D.pdf`, visited 2022-07-17.