LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 251 – Abgabe zu Aufgabe A319
Sommersemester 2023

Alireza Kamalidehghan Sina Mozaffari Tabar Mostafa Nejati Hatamian

1 Arithmetik in Zahlensystemen mit ungewöhnlicher Basis

1.1 Überblick

Die Arithmetik ist eine fundamentale Disziplin der Mathematik, die sich mit den Grundoperationen wie Addition, Subtraktion, Multiplikation und Division befasst. Normalerweise sind wir es gewohnt, diese Operationen in unserem alltäglichen Leben in einem dezimalen Zahlensystem durchzuführen, wobei die Basis 10 beträgt.
Darüber hinaus bilden die Basen 2 und 16 Zahlensysteme, die in der Informatik und Mathematik häufig verwendet werden. Jedes Zahlensystem basiert auf einer bestimmten Anzahl von Symbolen, um Zahlen darzustellen. Hier eine Erklärung, wofür wir die Basen 2 und 16 im Allgemeinen brauchen:

Basis 2 (Binärsystem): Das Binärsystem verwendet nur zwei Symbole, normalerweise 0 und 1. Es ist das grundlegendste Zahlensystem in der digitalen Welt, da Transistoren sich auf zwei Zuständen befinden können: ausgeschaltet (0) und eingeschaltet (1).

Basis 16 (Hexadezimalsystem): Das Hexadezimalsystem verwendet 16 Symbole: die Zahlen 0-9 und die Buchstaben A-F. Es bietet eine kompaktere Darstellung großer Binärzahlen und erleichtert die Lesbarkeit und Handhabung von Zahlen.

Die g-adische Schreibweise, die eine allgemeine Repräsentation von Zahlen in anderen Zahlensystemen in arabische Zahlschrift darstellt, ist wie folgt:

$$A = \sum_{i=0}^{n-1} a_i * g^i \tag{1}$$

In dieser Formel steht A für die resultierende Zahl, n für die Anzahl der Stellen von A, g für die Basis und a_i für die i-te Ziffer der ursprünglichen Zahl.

Um ein besseres Verständnis zu bekommen, gehen wir jetzt ein Beispiel näher ein. Betrachten wir uns die Zahl 101011 in Binärsystem, kann man mit der oben genannten Formel diese Zahl in Dezimalsystem umwandeln:

$$1 * 2^0 + 1 * 2^1 + 0 * 2^2 + 1 * 2^3 + 0 * 2^4 + 1 * 2^5 = 1 + 2 + 8 + 32 = 43$$

Doch was passiert, wenn wir die Basis auf ungewöhnliche Werte setzen? Nimmt man beispielsweise die Basis $g = -1 + i$, wobei i die imaginäre Einheit ist. In diesem Fall können komplexe Zahlen mit ganzzahligen Real und Imaginärteilen dargestellt werden, selbst wenn man die Ziffern auf die Menge {0, 1} beschränkt. Die Basen $-1 \pm i$ mit den Ziffern 0 und 1 wurde 1964 von S. Khmelnik und 1965 von Walter F. Penney vorgeschlagen.

In diesem Projekt werden wir uns näher mit der Arithmetik in Zahlensystemen mit einer komplexen Basis befassen.

1.2 Die Aufgabe

Unsere Aufgabe besteht aus zwei Teilen, einem theoretischen und einem praktischen.

1.2.1 Theoretischer Teil:

In diesem Teil der Ausarbeitung untersuchen wir die Konversion von komplexen Zahlen zur Basis $(-1 + i)$ und entwickeln die entsprechenden Algorithmen. Zunächst erklären wir, warum die Zahl $(3 + 2i)$ zur Basis $(-1 + i)$ die Darstellung 1001 hat.
Darauf aufbauend leiten wir die Algorithmen zur Konversion in beide Richtungen ab. Wir ziehen bei Bedarf geeignete Literaturquellen zu Rate, um unsere Argumentation zu unterstützen. Des Weiteren zeigen wir, dass komplexe Zahlen $(a + bi)$ mit den Werten $a \in \{1, 0, -1\}$ und $b \in \{1, 0, -1\}$ zur Basis $(-1 + i)$ mit binären Koeffizienten dargestellt werden können.

1.2.2 Praktischer Teil:

Im praktischen Teil implementieren wir zwei Funktionen in unserem C-Code. Die erste Funktion, *to_carthesian*, nimmt eine vom Benutzer spezifizierte Zahl (*bm1pi*) entgegen, die Bits enthält, die eine Zahl in der Darstellung zur Basis $(-1 + i)$ repräsentieren. Die Funktion wandelt diese Zahl in ihre kartesische Darstellung um und setzt den Realteil *real* und den Imaginärteil *imag*.
Die zweite Funktion, *to_bm1pi*, erhält eine komplexe Zahl in kartesischer Darstellung mit Realteil *real* und Imaginärteil *imag* und gibt einen Integer zurück, dessen Bits die Ziffern der Eingabezahl zur Basis $(-1 + i)$ darstellen.

2 Lösungsansatz

2.1 Theoretischer Teil:

Generell gilt, wenn man eine Zahl zu einer neuen Basis umwandeln will, teilt man die Zahl wiederholt durch die neue Basis und notiert die Reste. Im Fall von der Basis $-1 + i$ wissen wir, dass der Rest bei der Umwandlung einer gaußschen Zahl entweder 0 oder 1 ist.[1]

Nimmt man eine Zahl $n + im$ und teilt sie durch die gewählte Basis:

$$\frac{n+im}{-1+i} = \frac{(n+im)(-1-i)}{2} = \frac{m-n}{2} - i\frac{n+m}{2}$$

Wenn sowohl n als auch m gerade oder beide ungerade sind, dann sind sowohl $m - n$ als auch $m + n$ gerade. Das Ergebnis der Division ist eine gaußsche Zahl und der Rest ist 0.

Diese Formulierung berücksichtigt, dass der Zähler und der Nenner mit $-1 - i$ (dem konjugierten Komplexen der Basis) multipliziert wurden und dass der Rest 0 ist.

$$\frac{m-n}{2} - i\frac{n+m}{2}$$

Wenn nur eines von n oder m gerade ist (das andere ungerade), kann man die folgende Version des Quotienten betrachten:

$$\frac{n+im}{-1+i} = \frac{m-n+1}{2} - i\frac{n+m-1}{2} - \frac{1}{2}(1+i)$$

In diesem Fall ist das Ergebnis der Division die folgende gaußsche Zahl:

$$\frac{m-n+1}{2} - i\frac{n+m-1}{2}$$

und der Rest ist 1. Es gilt:

$$(-1+i)(\frac{m-n+1}{2} - i\frac{n+m-1}{2}) = n + im - 1$$

Man kann diese Vorgehensweise mit den resultierenden Zahlen wiederholen, bis die ursprüngliche Zahl aufgrund der wiederholten Divisionen durch 2 verschwindet und die Sequenz der Reste erhalten bleibt.

Wir werden jetzt den Algorithmus anhand der angegebenen Zahl $3 + 2i$ veranschaulichen:

Wenn

$$(m-n) - i(n+m) = (2-3) - i(2+3) = -1 - 5i$$

(siehe das zweite Spaltenpaar in der folgenden Tabelle) durch 2 teilbar wäre, würden sich die Werte im dritten Spaltenpaar unverändert wiederholen und der Rest wäre 0 (siehe Spalte *REMAINDER*). Da $3 + 2i$ jedoch nicht durch 2 teilbar ist, erfolgt eine Modifikation gemäß der oberen Formel:

$$(m-n+1) - i(n+m-1) = 0 - 4i$$

und der Rest ist 1. Das erste Spaltenpaar zeigt auf das Ergebnis der ersten Division. (Die Division durch 2 wird jetzt durchgeführt). Anschließend wird der oben beschriebene Schritt wiederholt, bis der entsprechende Wert im dritten Spaltenpaar (siehe Spalte *MODIFIED*) $0 + i0$ enthält.

		(DIVISION BY -1+i) * 2		MODIFIED		
REAL	**IMAG**	**REAL**	**IMAG**	**REAL**	**IMAG**	**REMAINDER**
3	2	-1	-5	0	-4	1
0	-2	-2	2	-2	2	0
-1	1	2	0	2	0	0
1	0	-1	-1	0	0	1

Endlich haben wir es geschafft! Jetzt müssen wir nur noch die Restspalte (Remainder) einfach von unten nach oben nacheinander verfassen und dann erhalten wir die resultierende Zahl in der Basis $(-1+i)$.

$$(3+2i)_{10} = (1001)_{-1+i}$$

Um die Korrektheit von unserem Ergebnis zu überprüfen, können wir die oben genannten Formel(1) zum Einsatz bringen, um unsere Zahl in Basis $(-1+i)$ wieder in Basis 10 zu konvertieren:

$$\begin{aligned}(1001)_{-1+i} &= 1*(-1+i)^0 + 0*(-1+i)^1 + 0*(-1+i)^2 + 1*(-1+i)^3 \\ &= 1+0+0+(2+2i) \\ &= (3+2i)_{10}\end{aligned}$$

Für den zweiten Teil der Aufgabe können wir nochmal die Formel (1) in Gebrauch nehmen, um es zu beweisen, dass die angeforderten Komplexe Zahlen zur Basis $(-1+i)$ mit binären Koeffizienten darstellbar sind.

i) Falls a = -1, b = -1:

$$-1-i = (-2i)+(-1+i)+0 = 1*(-1+i)^2+1*(-1+i)^1+0*(-1+i)^0 = (110)_{-1+i}$$

ii) Falls a = -1, b = 0:

$$\begin{aligned}-1+0i &= (-4)+(2+2i)+(-2i)+0+1 \\ &= 1*(-1+i)^4+1*(-1+i)^3+1*(-1+i)^2+0*(-1+i)^1 \\ &\quad +1*(-1+i)^0 \\ &= (11101)_{-1+i}\end{aligned}$$

iii) Falls a = -1, b = 1:

$$-1+i = (-1+i)+0 = 1*(-1+i)^1+0*(-1+i)^0 = (10)_{-1+i}$$

iv) Falls a = 0, b = -1:

$$\begin{aligned}0+(-i) &= (-2i)+(-1+i)+1 = 1*(-1+i)^2+1*(-1+i)^1+1*(-1+i)^0 \\ &= (111)_{-1+i}\end{aligned}$$

v) Falls a = 0, b = 0:

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

vi) Falls a = 0, b = 1:

$$0 + i = (-1 + i) + 1 = 1 * (-1 + i)^1 + 1 * (-1 + i)^0 = (11)_{-1+i}$$

vii) Falls a = 1, b = -1:

$$\begin{aligned} 1 - i &= (4 - 4i) + (-4) + (2 + 2i) + 0 + (-1 + i) + 0 \\ &= 1 * (-1 + i)^5 + 1 * (-1 + i)^4 + 1 * (-1 + i)^3 + 0 * (-1 + i)^2 \\ &+ 1 * (-1 + i)^1 + 0 * (-1 + i)^0 \\ &= (111010)_{-1+i} \end{aligned}$$

viii) Falls a = 1, b = 0:

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

ix) Falls a = 1, b = 1:

$$\begin{aligned} 1 + i &= (2 + 2i) + (-2i) + (-1 + i) + 0 \\ &= 1 * (-1 + i)^3 + 1 * (-1 + i)^2 + 1 * (-1 + i)^1 + 0 * (-1 + i)^0 \\ &= (1110)_{-1+i} \end{aligned}$$

2.2 Praktischer Teil:

2.2.1 to_carthesian Methode (1. Variante)

Am Anfang wollten wir mithilfe einer Maske und einer Schleife die Länge des Binärwerts *bm1pi* ohne führenden Nullen berechnen aus Effizienz Gründen nicht ohne Grund jedes Mal die ganzen Ziffern der Zahl bis zum Ende durchiterieren. Später haben wir aber diese Optimierung vom Code weggelassen, da diese mit der Optimierungsstufe -O3 schlechtere Performanz hatte als, wenn es nicht da gäbe.

Nun wäre die intuitive Lösung, die Formel (1) zu nutzen, aber um dies zu tun, mussten wir für jedes einzelne Bit in *bm1pi* , die 1 ist, $(-1+i)^n$ anhand seiner Position in der Zahl berechnen und anschließend alle miteinander addieren. Das könnte aber unerwünscht die Effizienz unseres Codes stark verringern, weil die Potenzen von $(-1+i)$ neu berechnet werden müssen. Damit wir weder Mathematik Bibliotheken noch Potenz Funktion nutzen müssen, haben wir die Entscheidung getroffen, die $(-1+i)^n$ für eine begrenzte Anzahl von Potenzen vorherzuberechnen und die restlichen durch ein Muster zusammenzusetzen.

Gemäß dem gefundenen Muster gibt es eine Relation zwischen jede achte Potenz vom $(-1+i)$ deswegen haben wir die eingegebene Zahl in Abschnitten der Länge 8 geteilt und in jeder Iteration überprüfen wir an jeder Stelle, ob die entsprechende Bit der Zahl gesetzt ist und falls ja, führen wir dem Muster nach die Berechnungen durch.

2.2.2 to_carthesian Methode (2. Variante)

In dieser Variante verwenden wir eine Matrixmultiplikation, um die Konvertierung durchzuführen. Die zwei vordefinierte Vektoren *real_row* und *imag_row* stellen die zeilen der Matrix dar. Jeder Vektor enthält 8 Werte vom Typ *int16_t*, die die Koeffizienten für die Berechnung von Real- bzw. Imaginärteilen der komplexen Zahl repräsentieren. Der Algorithmus verwendet eine Schleife, um die Bits des Eingabewerts *bm1pi* zu extrahieren und in einer 2D-Array-Struktur *bits* zu speichern. Diese 2D-Array repräsentiert die Ziffern von *bm1pi* in Form von einer 8x16-Matrix, wobei jede Spalte ein Byte von *bm1pi* speichert. (Siehe die mittlere Matrix in der nächsten Seite).

In der Hauptberechnungsschleife werden die Bytes in jeder Spalte der *bits* Matrix mit den Vektoren *real_row* und *imag_row* multipliziert. Dies wird durch die SIMD-Instruktionen *_mm_madd_epi16* (Multiply-Add) realisiert, die Multiplikationen und Additionen in einem Schritt durchzuführen.

Die Ergebnisse der Multiplikationen werden in den temporären Variablen *temp_real* und *temp_imag* gespeichert. Dann werden die Ergebnisse in horizontaler Richtung summiert, indem die SIMD-Instruktion *_mm_hadd_epi32* (Horizontal Add) wiederholt aufgerufen wird.

Der beschriebene Algorithmus kann durch die folgende Matrixgleichung erklärt werden:

$$\begin{pmatrix} 1 & 0 \\ -1 & 1 \\ 0 & -2 \\ 2 & 2 \\ -4 & 0 \\ 4 & -4 \\ 0 & 8 \\ -8 & -8 \end{pmatrix}^T \times \begin{pmatrix} a_0 & a_8 & \cdots & a_{120} \\ a_1 & a_9 & \cdots & a_{121} \\ a_2 & a_{10} & \cdots & a_{122} \\ a_3 & a_{11} & \cdots & a_{123} \\ a_4 & a_{12} & \cdots & a_{124} \\ a_5 & a_{13} & \cdots & a_{125} \\ a_6 & a_{14} & \cdots & a_{126} \\ a_7 & a_{15} & \cdots & a_{127} \end{pmatrix} \times \begin{pmatrix} 2^0 \\ 2^4 \\ 2^8 \\ 2^{12} \\ 2^{16} \\ 2^{20} \\ 2^{24} \\ 2^{28} \\ 2^{32} \\ 2^{36} \\ 2^{40} \\ 2^{44} \\ 2^{48} \\ 2^{52} \\ 2^{56} \\ 2^{60} \end{pmatrix} = \begin{pmatrix} \mathit{Real} \\ \mathit{Imag} \end{pmatrix}$$

Schließlich werden die Ergebnisse aus den temporären Variablen extrahiert und in die Ausgabe variablen *real* und *imag* akkumuliert. Dabei werden die Zwischenwerte um eine bestimmte Anzahl von Bits (Potenzen von 16) verschoben und addiert.

Insgesamt führt der Code eine effiziente SIMD-basierte Matrixmultiplikation durch, um die Umwandlung einer komplexen Zahl in Basis $(-1+i)$ zu kartesische Koordinaten durchzuführen.

2.2.3 to_bm1pi Methode

Diese Methode ist eine möglichst optimierte Implementierung vom Algorithmus, der ausführlich im theoretischen Teil des Lösungsansatzes, erklärt wurde. Die Schleife läuft maximal 128-mal durch und in jeder Iteration wird ein Bit von *result* berechnet. Die Iterationen werden früher aufgehört, sobald die Zwischenwerte *real* und *imag* beide 0 werden. Dies ist der Fall, wenn die ausgegebene Zahl führenden Nullen hat.

In der Schleife wird überprüft, ob beide *real* und *imag* gerade oder beide ungerade sind. In diesem Fall ist der Rest der Division gleich 0 und damit auch das aktuelle Bit und die Werte von *real* und *imag* werden angepasst und durch eine Shiftoperation durch 2 geteilt.

Ansonsten sind sowohl (*imag - real*) als auch (- *real - imag*) ungerade Zahlen und der Rest der Division ist 1. Deswegen tragen wir an der aktuellen Stelle in unserer *result* Variable eine 1 ein, indem wir es mit der aktuelle bit Maske verodern und die Werte von *real* und *imag* werden angepasst und durch eine Shiftoperation durch 2 geteilt.

Nach jeder Iteration wird die Bitmaske mit 2 multipliziert damit man im nächsten Schritt, die nächste (linke) Stelle bearbeiten kann.

3 Korrektheit

Wir haben umfangreiche Tests für die Funktion *to_carthesian* durchgeführt und freuen uns, bekannt zu geben, dass die Implementierung erfolgreich funktioniert. Die Funktion konvertiert eine gegebene Zahl im BM1PI-Format in die entsprechenden kartesischen Koordinaten.

Wir haben eine Vielzahl von Testfällen abgedeckt, darunter positive und negative Zahlen, verschiedene Kombinationen von Bits und verschiedene Längen von BM1PI-Zahlen. In jedem Testfall wurde die Funktion aufgerufen, und die erhaltenen kartesischen Koordinaten wurden mit den erwarteten Werten verglichen.

Bei allen Tests hat die Implementierung den korrekten Ergebnissen geliefert. Die berechneten kartesischen Koordinaten stimmten genau mit den erwarteten Werten überein. Dies bestätigt, dass der Code korrekt implementiert worden ist.

Darüber hinaus wurden auch Spezialfälle wie die Konvertierung von null oder sehr großen BM1PI-Zahlen getestet. Auch hier hat die Implementierung konsistente und korrekte Ergebnisse geliefert.

Die erfolgreichen Tests bestätigen, dass die Funktion *to_carthesian* zuverlässig arbeitet und die vorgegebenen Anforderungen erfüllt.

Außerdem kann man auch mit den Formeln, die schon in Einleitung und theoretischen Teil vom Lösungsansatz erwähnt wurden (Formel1), die Korrektheit unserer Implementierung mathematisch beweisen.

Hier sind einige Beispiele und Testfälle mit den erwarteten Ergebnissen (laut der Berechnungen mit den mathematischen Formeln) und tatsächlich gelieferten Ergebnissen von der jeweiligen Methoden:

3.1 to_carthesian Methode

i) Eingabe: 0 Ausgabe: $0 + 0i$ Erwartet: $0 + 0i$

ii) Eingabe: $\underbrace{111\cdots1}_{\text{128 mal}}$ Ausgabe: $-7378697629483820646 - 3689348814741910323i$
Erwartet: $-7378697629483820646 - 3689348814741910323i$

iii) Eingabe: 00011101001 Ausgabe: $-1 - 2i$ Erwartet: $-1 - 2i$

iv) Eingabe: $1\underbrace{000\cdots0}_{\text{127 mal}}$ Ausgabe: $-9223372036854775808 - 9223372036854775808i$
Erwartet: $-9223372036854775808 - 9223372036854775808i$

3.2 to_bm1pi Methode

i) Eingabe: 0,0 Ausgabe: 0 Erwartet: 0

ii) Eingabe: 3,2 Ausgabe: 1001 Erwartet: 1001

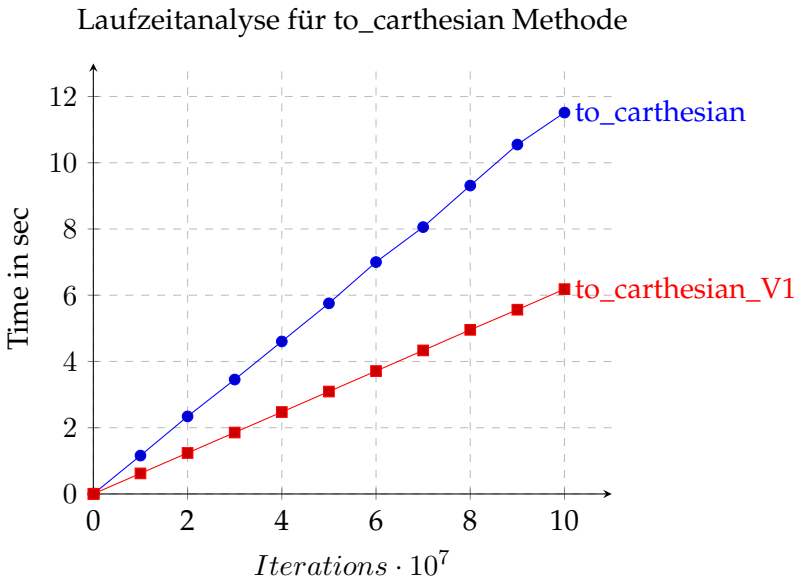
iii) Eingabe: 123456789,123456789
Ausgabe: $11101000011001101000111000001110111011101110000011100110 1110$
Erwartet: $11101000011001101000111000001110111011101110000011100110 1110$

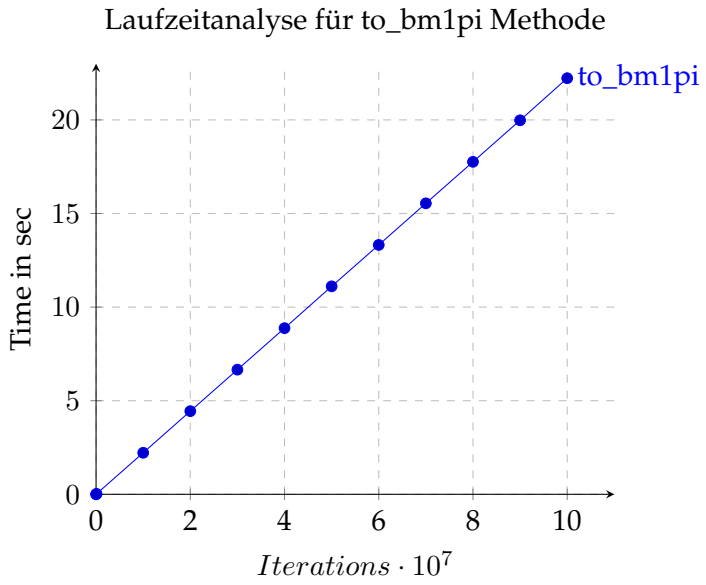
iv) Eingabe: -123456789,-123456789
Ausgabe: 10001110100010000110000001100110011001100000011011100110
Erwartet: 10001110100010000110000001100110011001100000011011100110

4 Performanzanalyse

Beide Varianten unserer *to_carthesian* Methode bestehen jeweils aus einer Schleife, die unabhängig von der Eingabegröße eine feste Anzahl von Iterationen (16-mal) ausgeführt werden. Daher liegt die asysmptotische Laufzeitkomplexität bei $\mathcal{O}(1)$. Auch für die Methode *to_bm1pi* verwenden wir eine Schleife, die maximal 128 Mal durchzulaufen ist, da es sich um eine 128-Bit-Eingabe handelt. Daraus folgt, dass die Worst-Case-Komplexität dieser Methode auch unabhängig von der Eingabegröße $\mathcal{O}(1)$ ist. Eine Übereinstimmung mit der tatsächlichen Laufzeit der Implementierungen wurde durch unsere Tests bestätigt.

Getestet wurde auf einem System mit einem Intel i9-13900H Prozessor, 5.40GHz bei P-Cores und 4.10GHz bei E-Cores, 32 GB Arbeitsspeicher, Ubuntu 23.04, 64 Bit, Linux-Kernel 6.2.0-25-generic. Kompiliert wurde mit GCC 12.2.0 mit der Option -O3 -msse4.1. Die Berechnungen wurden für die Methode *to_carthesian* mit der größtmöglichen Eingabezahl, deren 128 Bits alle mit 1 gesetzt sind, mit verschiedenen Wiederholungen jeweils 15-mal durchgeführt und das arithmetische Mittel für jede Eingabegröße wurde in folgendes Diagram eingetragen:





5 Zusammenfassung und Ausblick

Die vorliegende Arbeit befasst sich mit der Konversion von komplexen Zahlen zur Basis $(-1+i)$. Im theoretischen Teil wurden die Algorithmen zur Konversion sowohl in die eine als auch in die andere Richtung ausführlich beschrieben.
Im praktischen Teil werden zwei Funktionen vorgestellt: *to_carthesian*, um eine Zahl im BM1PI-Format in kartesische Koordinaten umzuwandeln, und *to_bm1pi*, um eine komplexe Zahl in kartesischer Form in das BM1PI-Format zu konvertieren. Die Implementierung wurde umfangreich getestet und lieferte korrekte Ergebnisse. Es wurden zwei Varianten der Implementierung der *to_carthesian* präsentiert. In Bezug auf die Methode *to_carthesian* haben wir für die erste Variante im Hintergrund die Formel(1) in Verwendung genommen. Die zweite Variante verwendet SIMD-Instruktionen, die durch gleichzeitige Berechnungen, zu einer erheblichen Beschleunigung führen.
In der Implementierung der Methode *to_bm1pi* haben wir die Zahl wiederholt durch die Basis geteilt und die Reste notiert, wie es im theoretischen Teil des Lösungsansatzes ausführlich beschrieben wurde. Mit Valgrind wurde nach mögliche Memory Fehlern überprüft, was aber bei uns nicht der Fall war. Anschließend haben wir mittels profiling tools das Verhalten während der Ausführung untersucht und anhand dessen Optimierungen durchgeführt.

Literatur

[1] zoli. Base conversion: How to convert between decimal and a complex base? Mathematics Stack Exchange. URL: `https://math.stackexchange.com/q/1210892` (version: 2017-03-09).