LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Gruppe 134 – Abgabe zu Aufgabe A319

Sommersemester 2023

Chen Yang Qichen Liu Tanmay Amarendra Deshpande

1 Einleitung

1.1 Arithmetik in Zahlensystemen mit komplexer Basis

Wir kennen die g-adische Schreibweise einer Zahl A als Abfolge von Ziffern a_i mit ihrer Position entsprenchender Wertigkeit gemäß g:

$$A = \sum_{i=0}^{n} a_i \cdot g^i$$

In der Aufgabe beschäftigen wir uns mit Arithmetik in Zahlensystemen, die auf einer komplexen Basis $g = -1 + i$ mit $i^2 = -1$ basieren.
Somit[4] können alle komplexen Zahlen mit ganzen Real- und Imaginärteilen dargestellt werden, und das sogar, wenn man a_i als Elemente der Menge $\{0, 1\}$ festsetzt.
Das Letztere bezieht sich auf die übliche binäre Zifferndarstellung in der Informatikwelt.
Wir entwerfen daher einen Rechner, der zwischen der üblichen kartesischen Darstellung einer komplexen Zahl $a + bi$ und ihrer Darstellung zur Basis $-1 + i$ konvertieren kann.

1.2 Aufgabenstellung

Die Aufgabe lässt sich in die Bereiche Konzeption (theoretisch) und Implementierung (praktisch) aufteilen. Antworten auf konzeptionelle Fragen werden an den passenden Stellen in der Ausarbeitung erscheinen. Für die Implementierung sind die folgenden zwei Signaturen in der Programmiersprache C vorgegeben.

```
1 void to_carthesian(unsigned __int128 bm1pi, __int128* real,
      __int128* imag);
2 unsigned __int128 to_bm1pi(__int128 real, __int128 imag);
```

Die Funktion `void to_carthesian` konvertiert eine komplexe Zahl `unsigned __int128 bm1pi` zur Basis $-1 + i$ in ihre übliche kartesische Darstellung. Der Realteil wird an die Adresse `__int128 *real` gespeichert und der Imaginärteil an die Adresse `__int128 *imag`.
Die Funktion `unsigned __int128 to_bm1pi` konvertiert die übliche kartesische Darstellung einer komplexen Zahl mit dem Realteil `__int128 real` und dem Imaginärteil `__int128 imag` in ihre Darstellung zur Basis $-1 + i$. Das Ergebnis wird als `unsigned __int128` zurückgegeben.
Es ist wichtig anzumerken, dass in der Implementierung nur die vier Grundrechenarten $(+, -, \times, \div)$ erlaubt sind.

2 Lösungsansatz

2.1 Umrechnung von der kartesischen Form in die Basis $-1+i$[1]

Eine Dezimalzahl lässt sich in eine andere Basis konvertieren, indem man die Zahl ständig durch den Wert der neuen Basis dividiert, bis man auf die 0 kommt und bei jedem Divisionsschritt den Rest notiert. Man baut dann die neue Darstellung auf, indem man die Reste der Divisionen in umgekehrter Reihenfolge aneinanderhängt. Als Beispiel rechnen wir die Dezimalzahl 9 in das Binärsystem um. Ergebnis: $9_{10} = 1001_2$.

Dividend	Divisor	Quotient	Rest
9	2	4	1
4	2	2	0
2	2	1	0
1	2	0	1

Analog lässt sich die Zahl auch in die Basis $-1+i$ konvertieren. Sei die komplexe Zahl in Dezimalsystem $a+bi$ wobei a den Realteil und b den Imaginärteil sind. Nach einer Division durch $-1+i$ erhalten wir:

$$\frac{a+bi}{-1+i} = \frac{b-a}{2} - \frac{a+b}{2} \cdot i$$

Fall 1: a und b beide gerade bzw. beide ungerade:
Sowohl $b-a$ als auch $a+b$ sind durch 2 teilbar. Daher erhalten wir den Quotienten $\frac{b-a}{2} - \frac{(a+b)i}{2}$ mit einem Rest von 0.
Fall 2: a gerade und b sind ungerade oder umgekehrt:
Sowohl $b-a$ als auch $a+b$ ungerade. Die Quotienten lassen sich aber wie folgt umformulieren, somit haben wir den Rest von 1 mit dem Quotienten $\frac{b-a+1}{2} - \frac{a+b-1}{2} \cdot i$

$$\frac{b-a}{2} - \frac{a+b}{2} \cdot i = \frac{b-a+1}{2} - \frac{a+b-1}{2} \cdot i - \frac{1+i}{2}$$

Daraus folgt, dass der Rest entweder 0 oder 1 ist. Somit können komplexe Zahlen als einzelner Bitstring dargestellt werden, anstatt den Real- und Imaginärteil getrennt zu schreiben. Bei jedem Schritt der Division wenden wir die entsprechenden Formeln für den Quotienten an, je nachdem ob beide Teile gerade oder ungerade sind oder unterschiedliche Teilbarkeiten durch 2 aufweisen, und setzen den Rest auf 0 bzw. 1. Der Algorithmus wird beendet, wenn sowohl der Realteil als auch der Imaginärteil 0 sind. Als Beispiel rechnen wir $3-2i$ in Basis $-1+i$ um:

Dividend (Real)	Dividend(Imag)	Divisor	Quo. (Real)	Quo.(Imag)	Rest
3	-2	$-1+i$	-2	0	1
-2	0	$-1+i$	1	1	0
1	1	$-1+i$	0	-1	0
-1	0	$-1+i$	0	1	1
0	1	$-1+i$	1	0	1
1	0	$-1+i$	0	0	1

Als Ergebnis lesen wir die Reste in umgekehrter Reihenfolge: $(111001)_{-1+i}$

2.1.1 to_bm1pi_V1: Naive Implementierung

Die naive Implementierung übersetzt den Algorithmus aus dem vorherigen Abschnitt [1] in C-Code mithilfe einer while-Schleife. Der Real- und Imaginärteil werden vor der nächsten Iteration gemäß den Formeln aktualisiert, bis beide Teile den Wert 0 erreichen. Das Setzen des Bitstrings (abhängig vom Rest in jeder Iteration) erfolgt durch Verwendung einer Maske, die anfangs den Wert 1 hat und bei jeder Iteration um eine Position nach links verschoben wird. Durch Veroderung (OR) mit dem Bitstring kann die entsprechende Position auf den Wert 1 gesetzt werden. Die Variable `shift` zählt, um wie viele Stellen die Maske nach links verschoben wurde. Falls die Maske mehr als 128 Mal verschoben wurde, bedeutet dies, dass die Zahl nicht in 128-Bit darstellbar ist. In diesem Fall wird eine Fehlermeldung mit `perror()` ausgegeben und das Programm mit `EXIT_FAILURE` abgebrochen.

Algorithm 1 Pseudocode für unsigned __int128 to_bm1pi(__int128 real, __int128 imag)

$bitstring \leftarrow 0$
$mask \leftarrow 1$
$shift \leftarrow 0$
while !($imag == 0$ and $real == 0$) **do**
  if ($real$ is even and $imag$ is even) or ($real$ is odd and $imag$ is odd) **then**
    $real \leftarrow (imag - real)/2$
    $imag \leftarrow (-1 \times (imag + real))/2$
  else
    $real \leftarrow (imag - real + 1)/2$
    $imag \leftarrow (-1 \times (real + imag - 1))/2$
    $bitstring \leftarrow bitstring \vee mask$
  end if
  $mask \leftarrow mask \ll 1$
  $shift \leftarrow shift + 1$
  if $shift > 128$ **then**
    $error : real + imag * i \ \ does \ \ not \ \ fit \ \ in \ \ 128 \ \ bit$
  end if
end while
return $bitstring$

2.1.2 to_bm1pi: Multidivision Optimierung

Da der Algorithmus sequentiell ist, ist es schwer mittels SIMD parallelisierbar. Die Werte, die bei jeder Iteration verarbeitet werden sollen, hängen von den Werten ab, die in der vorherigen Iteration berechnet wurden. Daher müssen andere Optimierungsstrategien in Betracht gezogen werden.
Diese Funktion versucht, die naive Implementierung zu beschleunigen, indem sie überprüft, ob die gegebene Dezimalzahl und die Zwischenergebnisse des vorherigen Algo-

rithmus durch $(-1+i)^2$ oder $(-1+i)^3$ teilbar sind. Wenn dies der Fall ist, überspringt die Funktion die Zwischenschritte und berechnet das Ergebnis direkt.
Zum Beispiel gilt $(2+2i) = (-1+i)^3$. Für eine Zahl $(a+bi)$, die durch $(2+2i)$ teilbar ist, kann das Ergebnis von $\frac{a+bi}{2+2i}$ wie folgt geschrieben werden:

$$\frac{a+bi}{2+2i} = \frac{2(a+b)+2(b-a)i}{8} = \frac{a+b}{4} + \frac{b-a}{4} \cdot i$$

Der Realteil des Ergebnisses wird zu $\frac{a+b}{4}$ und der Imaginärteil wird zu $\frac{b-a}{4}$. Da wir durch $(-1+i)^3$ dividiert haben, können wir drei Schritte überspringen und an den nächsten drei Stellen im Bitstring den Rest als 0 setzen, da die Zwischenergebnisse immer durch $(-1+i)$ teilbar sind.
Analog können wir vorgehen, wenn die Dezimalzahl $-2i = (-1+i)^2$ teilbar ist, und 2 Schritte überspringen. Dies führt zu einer Beschleunigung, insbesondere in Fällen, in denen die Repräsentation zur Basis $(-1+i)$ viele aufeinanderfolgende Nullen enthält.

2.2 Umrechnung von der Basis $-1+i$ in die kartesische Form

Eine Zahl zur Basis $(-1+i)$ lässt sich wie folgt in kartesiche Darstellung konvertieren:

$$(101)_{-1+i} = 1 \cdot (-1+i)^0 + 0 \cdot (-1+i)^1 + 1 \cdot (-1+i)^2 = 1 + 0 + (-2i) = 1 - 2i$$

2.2.1 to_carthesian_V2: Naive Implementierung

Die Funktion `void to_carthesian_V2` nimmt eine Zahl zur Basis $-1+i$ und konvertiert sie in Dezimalsystem mit getrennten Real- und Imaginärteil.

Algorithm 2 to_carthesian Pseudocode

```
*real ← 0
*imag ← 0
mask1 ← 1
base ← {−1, 1}
tmp ← struct complex_number
for i ← 0 to 127 do
    digit ← bm1pi & mask1
    bm1pi ← bm1pi ≫ 1
    if ¬digit then
        continue
    end if
    exponent_of_base(i, &base, &tmp)
    *real ← *real + tmp.real
    *imag ← *imag + tmp.imag
end for
```

Variable `tmp` wird verwendet, um die Summe der Real- und Imaginärteile zu speichern,

während `base` als Instanz von `struct complex_number` definiert wird, was später nützlich sein kann. Bei jeder Iteration wird das entsprechende Bit aus `bm1pi` mithilfe einer Verundung (AND) mit der Maske extrahiert und in `digit` gespeichert. Wenn `digit` den Wert 0 hat, wird die Iteration übersprungen. Andernfalls wird die Potenz von $(-1+i)$ mit dem entsprechenden Bitindex als Exponent berechnet. Anschließend werden der Realteil und der Imaginärteil dieser Potenz auf `tmp` addiert.
Die Funktion `complex_mul_base()` implementiert die Multiplikation einer komplexen Zahlen mit $(-1+i)$. Die Funktion `exponent_of_base()` verwendet `complex_mul()`, um eine bestimmte Basis n-Mal mit sich selbst in einer for-Schleife zu multiplizieren. Dadurch kann die Summe der Potenzen berechnet werden, nur unter Verwendung der vier Grundrechenarten.

2.2.2 to_carthesian_V1: Lookup-Tabelle

Es ist nicht effizient genug, wenn man bis zu 128 Potenzen der Basis $(-1+i)$ in der Ausführung des Programms berechnet. Eine Lookup-Tabelle (ein Array von `struct complex_number`) eignet sich gut zur Beschleunigung der Performanz von `void to_carthesian`, weil die maximale Größe der Darstellung der Zahl zur Basis $(-1+i)$ bereits bekannt ist. Wir haben eine Lookup-Tabelle erstellt, in der die ersten 128 Potenzen (0-127) der Basis $(-1+i)$ als `struct complex_number` gespeichert sind. Dadurch führen wir lediglich einen Random-Access auf ein Array durch, anstatt Multiplikationen während der Behandlung jedes Bits durchzuführen.

```
1 struct complex_number powers[] = {{1, 0},{-1, 1},
2     {0, -2},..... {-9.223372036854775808e+18, -9.223372036854775808e
          +18}};
```

Algorithm 3 Pseudocode: to_carthesian Lookup Tabelle

for $i = 0$ to 127 **do**
  $digit \leftarrow bm1pi \;\&\; mask1$
  $bm1pi \leftarrow bm1pi \gg 1$
  if $\neg digit$ **then**
    continue
  end if
  $*real \leftarrow *real + powers[i].real$
  $*imag \leftarrow *imag + powers[i].imag$
end for

2.2.3 to_carthesian : Lookup-Tabelle SIMD

Dies ist eher keine Optimierung, sondern eine alternative Version der vorheringen Implementierung. Fast alle Potenzen von $(-1+i)$ von 0 bis 127 lassen sich in 64-Bit darstellen, außer $(-1+i)^{126}$ und $(-1+i)^{127}$. Für die ersten 126 Potenzen können der Real-

und der Imaginärteil einer Potenz aus dem Array `powers64` (definiert in `base_powers.c`) mittels SIMD-Intrinsics gleichzeitig in einem 128-Bit-Register geladen werden. Der Realteil wird in den oberen 64 Bit des Registers gespeichert und der Imaginärteil in den unteren 64 Bit. Dann kann mit dem Befehl `_mm_add_epi64()` gleichzeitig eine Addition der Real- und Imaginärteile einer Potenz durchgeführt werden, wodurch im Vergleich zur vorherigen Implementierung eine Addition eingespart wird. Die Ladeoperation der beiden Teile erfolgt ebenfalls gleichzeitig. Für die letzten zwei Potenzen wird eine normale Addition durchgeführt.
Wie bereits angedeutet, ist der dadurch erhaltene Performanzgewinn relativ gering. Das eigentliche Aufaddieren mehrerer Potenzen lässt sich aufgrund der Größe des Datentyps schwer parallelisieren, weshalb SIMD nur eine geringfügige Beschleunigung bringt.

3 Korrektheit

3.1 Programmausgabe

Wir arbeiten mit den Datentypen `__int128` und `unsigned __int128`, für die in C keine integrierten Funktionen vorhanden sind. Daher haben wir eigene Hilfsfunktionen in `utils.c` implementiert, um mit diesen Zahlen zu arbeiten und sie auszugeben. [2]

```
The default iteration of 1 is used
time average: 0.000007
(-595) + (-1282)i
```

Listing 1: Ausgabe mit Command ./cbns -V 0 -B 123456789. Implementierung v0 wird hierbei verwendet und benchmark ist aktiviert mit 1 Iteration.

```
The default iteration of 1 is used
Time average: 0.000003
Result as decimal:
215436046512475
Result as binary:
1100001111110000000111010000011000110001011011
```

Listing 2: Ausgabe mit Command ./cbns -V 1 -B -8917234,786123. Implementierung v1 wird hierbei verwendet und benchmark ist aktiviert mit 1 Iteration.

3.2 Test

3.2.1 Vordefinierte Tests mit ausgesuchten Edge Cases

Zunächst überprüfen wir die Korrektheit aller unserer Implementierungen anhand von Testfällen, die wichtige Randfälle und Grenzen abdecken. Da wir mit Bitstrings zur Basis $-1 + i$ arbeiten und diese eine maximale Länge von 128 Bit haben, existieren obere und untere Schranken für den Real- und Imaginärteil.
Um sie zu finden, laufen wir unsere Lookup-Tabelle durch und iterieren dabei mithilfe einer for-Schleife durch alle Potenzen. Während dieser Iteration berechnen wir vier Grenzwerte: REAL_MAX (eine 128-Bit-Zahl mit maximalem Realteil), IMAG_MAX (eine

128-Bit-Zahl mit maximalem Imaginärteil), REAL_MIN (eine 128-Bit-Zahl mit minimalem Realteil) und IMAG_MIN (eine 128-Bit-Zahl mit minimalem Imaginärteil).
Zuerst testen wir alle Implementierungen von `to_carthesian` mit Bitstrings unterschiedlicher Längen. Ebenso werden alle Implementierungen von `to_bm1pi` getestet, um sicherzustellen, dass sie die erwarteten Bitstrings als Ergebnis liefern. Wir haben Bitstrings gewählt, deren Längen Vielfache von 8 sind (also 8, 16, ... 128), um einen breiten Bereich von Zahlen abzudecken.
Zusätzlich überprüfen wir vordefinierte Tests, die wichtige Grenzfälle und Randfälle abdecken. Die Tests umfassen zum Beispiel die Zahl 0 und die zuvor gefundenen Grenzwerte. Das Bestehen der vordefinierten Tests zeigt, dass die Funktionen für verschiedene Bitlängen und wichtige Randfälle funktionieren. Dadurch erhalten wir eine vertrauenswürdige Grundlage, um die Korrektheit der Implementierungen zu bestätigen.

3.2.2 Quervergleich mit randomisierten Eingaben

Mit vordefinierten Tests kann man spezifische Fälle überprüfen. Wenn jedoch die Korrektheit über einen breiten Bereich getestet werden soll, ist es nicht realistisch, eine große Anzahl von Tests manuell zu generieren und durchzuführen. Aus diesem Grund haben wir randomisierte Tests implementiert. Wir beginnen mit einem zufälligen Startwert, und in jeder Iteration führen wir die Funktion `to_carthesian()` mit einer zufälligen Eingabe aus. Das Ergebnis wird dann als Eingabe für die Funktion `to_bm1pi()` verwendet, und wir überprüfen, ob das Ergebnis von `to_bm1pi()` mit der ursprünglichen Eingabe übereinstimmt. Man kann alle Tests ganz bequem mit Option `-t` laufen lassen und sehen, wie wir den Test aufgebaut und gestaltet haben.

Algorithm 4 Vereinfachter code für randomisierte Tests

for $i = rs$ to $rs + iter$ **do**
  $to\_carthesian(i, real, imag)$
  $complex \leftarrow to\_bm1pi(real, imag)$
  $failed \leftarrow complex \neq i$
end for

4 Performanzanalyse

4.1 Testumgebung

Die Performanzanalyse wurde auf einem Computer mit der folgenden Specifikationen durchgeführt:

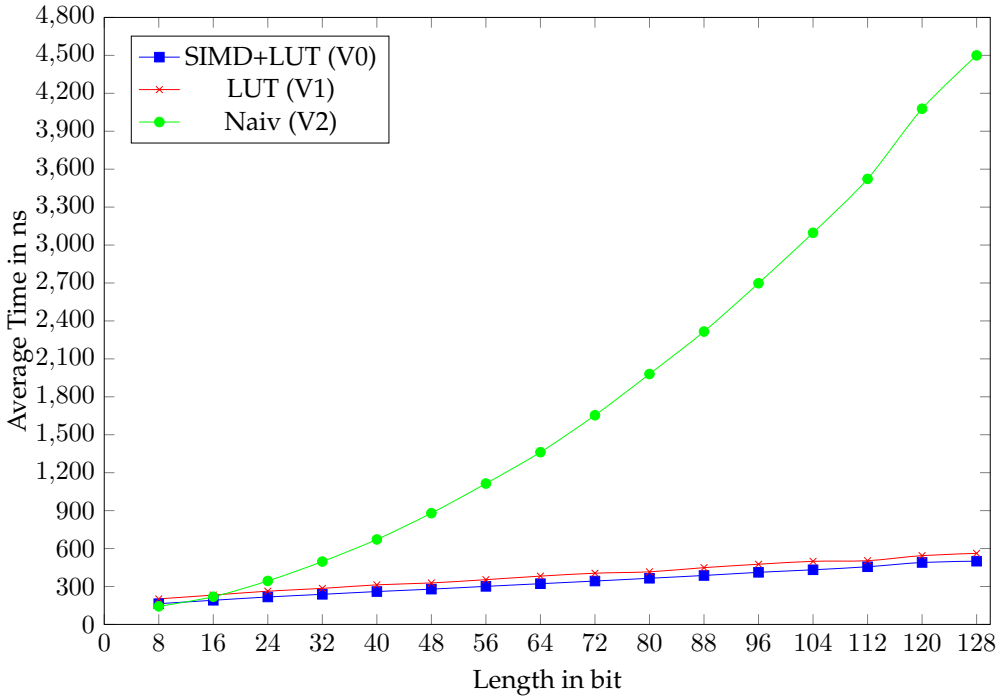
- OS: Fedora Linux 37 (Workstation Edition) x86_64
- Kernel: 6.0.18-300.fc37.x86_64
- Prozessor: Intel i5-9400F (6) @ 4.100GHz

- RAM: 8G DDR4 2667Hz

Das Programm wurde mit Compiler gcc version 12.3.1 und mit Optimierungsstufe -O2 kompiliert.

4.2 to_carthesian

Um eine allgemein schnellere Optimierung für alle möglichen Eingaben zu erreichen, haben wir die Leistungsfähigkeit wie folgt getestet: Wir haben das Programm mit Eingaben unterschiedlicher Längen zwischen 8 und 128 getestet, wobei der Schritt zwischen den Längen 8 betrug. Für jede Bitlänge wurde das Programm 1.000.000 Mal ausgeführt, wobei die Zahl in jeder Iteration zufällig generiert wurde. Das Ergebnis wird im nachfolgenden Liniendiagramm dargestellt:



LUT und SIMD Optimierung Aus dem Graphen geht hervor, dass die Hauptimplementierung(SIMD+LUT) die schnellste ist und eine lineare Laufzeit mit zunehmender Bitlänge der Eingabe aufweist. Die reine LUT-Implementierung zeigt einen leichten Anstiegstrend, was auf eine lineare Zeitkomplexität hinweist. Der Anstieg ist nahezu identisch mit der Hauptimplementierung, jedoch ist die Laufzeit im Allgemeinen etwas langsamer. In beiden Fällen wird hauptsächlich eine For-Schleife mit 126 Iterationen ausgeführt. In jeder Iteration werden einige konstante Operationen wie logische UND-, Bit-Shift- und Vergleichsoperationen mit 0 durchgeführt. Zusätzlich gibt es Additions- und Load-Operationen, wenn das entsprechende Bit gesetzt ist. Je länger die Zahl, desto

mehr Bits werden gesetzt. Beide Algorithmen haben eine lineare Laufzeitkomplexität:

$$V0 : O(a_0 \cdot n + c \cdot 126) = O(a_0 \cdot n), \;\; V1 : O(a_1 \cdot n + c \cdot 126) = O(a_1 \cdot n), \;\; a_0 < a_1$$

wobei c die Zeit für die Operationen entspricht, die in jeder Iteration ausgeführt werden müssen, und a_0 und a_1 die Zeit für die Operationen, wenn das entsprechende Bit gesetzt ist. n ist der Anzahl der gesetzten Bits. Die letzten Iterationen werden speziell behandelt. Ursprünglich hatten wir eine Beschleunigung von 2x durch die SIMD-Optimierung im Vergleich zur reinen LUT-Implementierung erwartet, da wir die beiden Additionen von Real- und Imaginärteil mit der `_mm_add_epi64`-Instruktion auf eine Addition reduziert hatten. Allerdings muss immer noch in jeder Iteration der Offset zusätzlich mit einer Addition berechnet werden, was insgesamt ebenfalls 2 Additionen entspricht. Mit der Verwendung von `_mm_loadu_si128` haben wir jedoch immer noch eine Load-Operation weniger in jeder Iteration im Vergleich zur reinen LUT-Implementierung. Dies ist auch der Grund ist, warum $a_0 < a_1$. Bei der reinen LUT-Implementierung müssen Real- und Imaginärteil separat mit Array-Zugriffen geladen werden, was zu einer etwas längeren Laufzeit bei jeder Bitlänge führt.

Naive Implementierung Die Ausführungszeit der naiven Implementierung steigt mit der steigenden Bitlänge quadratisch an. In der Hauptschleife werden auch einige Grundoperationen durchgeführt, und wenn das Bit gesetzt ist, werden zusätzlich 2 Additionen und die Funktion `exponent_of_base` aufgerufen, die $(-1 + i)^n$ berechnet. Die Funktion `exponent_of_base` selbst führt hauptsächlich eine For-Schleife aus, die eine lineare Laufzeit hat. Daher liegt die Laufzeit der V2 am Ende in der Komplexitätsklasse:

$$O(128 \cdot c + \sum_{n \in I} n) = O(n^2), \;\; I \subseteq \{0, 1, 2, ..., 127\}$$

was einer quadratischen Komplexität entspricht. I ist die Menge der Indizes der gesetzten Bits und c ist die Zeit für die Operationen in jeder Iteration.

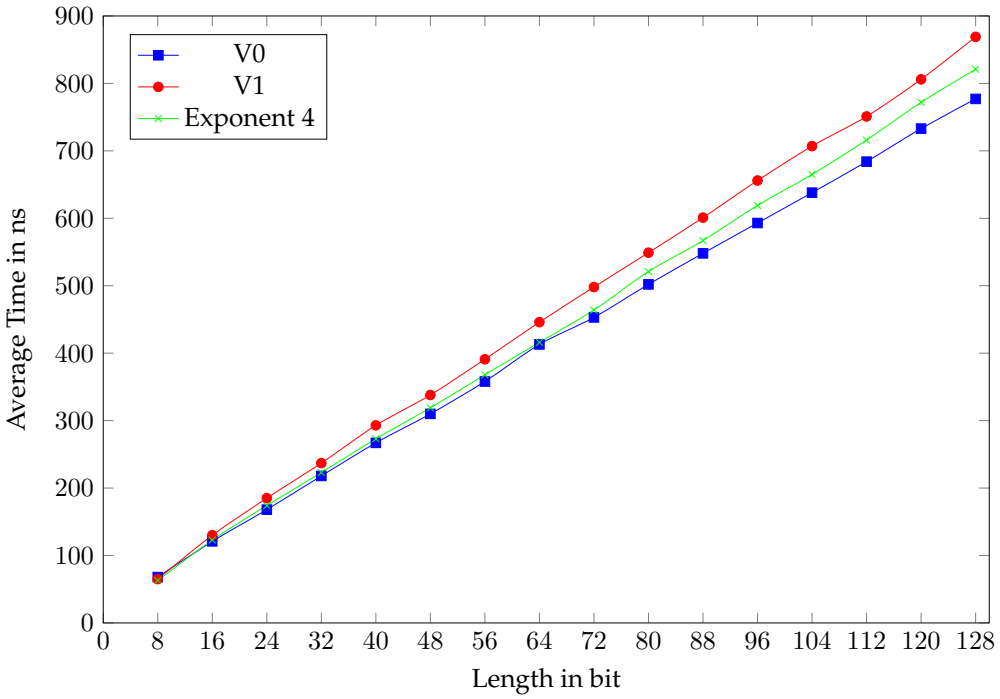
4.3 to_bm1pi

Analog zu `to_carthesian`, haben wir zufällige komplexe Zahlen als Eingabe generiert, deren binäre Darstellung Bitlängen zwischen 8 und 128 sind, ebenso mit einem Abstand von 8. Die Anzahl der Ausführungsiterationen bleibt ebenfalls konstant bei 1.000.000.
Die Laufzeit beider Implementierungen steigt mit zunehmender Bitlänge linear an, wobei V1 eine etwas größere Steigung aufweist im Vergleich zu V0. V0 verwendet eine Multi-Division-Optimierung, die mit steigender Bitlänge allmählich mehr Zeit einspart. Beide Implementierungen führen hauptsächlich eine `while`-Schleife durch, in der nur die vier Grundrechenarten durchgeführt werden. Daher liegen V0 und V1 beide in der lineare Laufzeitklasse:

$$O(a \cdot n), \;\; O(b \cdot n) \;\; mit \;\; a > b$$

wobei a und b die Zeit für die eigentliche Berechnung repräsentieren und n die Bitlänge der binären Darstellung der komplexen Zahl beschreibt. Das Ergebnis wird im folgenden

Diagramm dargestellt:



Die Optimierung zielt darauf ab, die Berechnung für aufeinanderfolgende 0en zu überspringen bzw. nur einmal durchzuführen, wenn die Zahl durch $(-1+i)^2$ oder $(-1+i)^3$ ohne Rest teilbar ist. Je länger die Bitlänge der binären Darstellung der komplexen Zahl, desto mehr aufeinanderfolgende Blöcke von Nullen es gibt. Daher spart V0 mit zunehmender Bitlänge allmählich mehr Zeit. Wir haben die Exponenten 2 und 3 von $(-1+i)$ gewählt, da sie die wahrscheinlichsten Längen für aufeinanderfolgende Blöcke von Nullen sind. Größere Exponenten könnten theoretisch auch verwendet werden, was jedoch zu mehr Modulo- und Vergleichsoperationen führen würde und die Gesamtlaufzeit verlangsamen würde, wie im Diagramm dargestellt. Die folgende Tablle zeigt 2 Fälle, wobei unsere Optimierung besonders gut und sogar langsamer ist:

real	imag	time(V0)	time(V1)	bm1pi
4096	0	63ns	139ns	100000000000000000000000
-1638	-819	162ns	155ns	111111111111111111111111

Im ersten Fall hat die zu konvertierende komplexe Zahl $4096+0i$ eine binäre Darstellung, bei der nur das signifikante Bit 1 ist und alle anderen Bits 0 sind. Dabei überspringt V0 die sukzessiven Nullen sehr schnell und weist im Vergleich zur naiven Implementierung eine Beschleunigung von mehr als 2x auf.
Im zweiten Fall gibt es keine aufeinanderfolgenden Nullen, daher ist V0 sogar langsamer, da in jeder Iteration zusätzlich zwei `if`-Konditionen überprüft werden müssen.

4.3.1 Einfluss des Caches auf die Leistung

Der Cache spielt eine wichtige Rolle, insbesondere wenn das Programm mit der Option `-B` und einer großen Anzahl an Iterationen ausgeführt wird. Die durchschnittliche Laufzeit kann sich erheblich unterscheiden, aufgrund der unterschiedlichen Anzahl von Cache-Misses. Als Beispiel: Bei der Ausführung des Befehls `./cbns 52045582664164951 -B1000000`, wobei 52045582664164951 eine 32-Bit-lange Zahl ist, kann der Cache einen signifikanten Einfluss auf die Laufzeit haben.

Eingabe	Cache-Misses	Zeit
52045582664164951	6	135ns
zufälligen 56-bit Zahlen	340	303ns

Im ersten Fall werden die Daten, die von der LUT (Lookup-Tabelle) geladen wurden, im Cache gespeichert und in den nächsten Iterationen wiederverwendet, da die Eingabe unverändert bleibt. Im Gegensatz dazu ist die Wahrscheinlichkeit der Wiederverwendung im Fall zufälliger Eingaben deutlich geringer. Dies liegt daran, dass die Indizes der gesetzten Bits jedes Mal unterschiedlich sind, was zu einer erhöhten Anzahl von Cache-Misses führt. Bei der Funktion `to_bm1pi()` besteht eine ähnliche Situation.

5 Zusammenfassung und Ausblick

Unsere allgemeine Vorgehensweise lässt sich wie folgt beschreiben: Zuerst implementieren wir eine naive Lösung, dann versuchen wir, die einzelnen Rechenschritte zu parallelisieren.
Genauer betrachtet haben wir in `void to_carthesian` zuerst versucht, durch die Bitstellen zu iterieren und die entsprechende Potenz von$-1+i$ zu berechnen. Dieses Zwischenergebnis haben wir dann zum Endergebnis addiert. Zur Optimierung haben wir die Potenzen von $-1+i$ bis zu 128 Stellen im Voraus berechnet und in eine Lookup-Tabelle eingetragen. Dadurch konnten wir teure Multiplikationen vermeiden, indem wir nicht jedes Mal die Potenz berechnen mussten. Am Ende haben wir auch 128-Bit-SSE-Register verwendet, um den Real- und Imaginärteil gleichzeitig zu laden. Dies ermöglicht es uns, bis auf zwei Real- oder Imaginärteile, die nicht mehr in einem 64-Bit-Register passen, effizient zu verarbeiten. Dadurch sparen wir einen Ladeschritt.
In `unsigned __int128 to_bm1pi` haben wir zuerst versucht, analog zu den üblichen Basiskonvertierungen, die komplexe Zahl Schritt für Schritt durch $-1+i$ zu dividieren, bis es keinen Rest mehr gibt. Zur Optimierung haben wir die Zahl in jedem Schritt durch die mehrfach Potenz von $-1+i$ dividiert, um unnötige Divisionsschritte zu sparen.
Eine weitere Vorgehensweise[3] ist die Konvertierung kartesischer Zahlen in Basis -4 und anschließendes Konvertieren in Basis $(-1+i)$. Diese Methode braucht, dass man auch Arithmetik in Basis $(-1+i)$ implementiert, und daher auch Rechnen mit $(-1+i)$ Basis Bitstrings ermöglicht. Diese ist allerdings von uns nicht implementiert und nur als Ausblick erwähnt.

Literatur

[1] `https://math.stackexchange.com/questions/1209190/base-conversion-how-to-convert-between-decimal-and-a-complex-base`, visited 2023-6-29.

[2] `https://github.com/DoctorManhattan123/int128/blob/main/int128_lib.h`, visited 2023-6-29.

[3] T. Jamil. The complex binary number system. *IEEE Potentials*, 20:39–41, 2002. `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=983342`, visited 2023-6-29.

[4] Walter Penney. A "binary" system for complex numbers. *J. ACM*, 12:247–248, 1965. `https://dl.acm.org/doi/10.1145/321264.321274`, visited 2023-6-29.