LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Praktikum Rechnerarchitektur

Gruppe 110 – Abgabe zu Aufgabe A212

Wintersemester 2019/20

Shutao Shen Zhiyi Pan Yujie Zhang

1 Einleitung

In dieser Ausarbeitung konzentrieren wir uns auf ein Multicorn-Fraktal, insbesondere das Tricorn-Fraktal. Die Besonderheit des Tricorn-Fraktals ist, dass die Gebilde oder Muster des Tricorn-Fraktals einen hohen Grad von Skaleninvarianz bzw. Selbstähnlichkeit aufweisen.[4] [5] Unsere Aufgabe ist es, ein Tricorn-Fractal zu visualisieren. Dabei müssen 5 vom Benutzer eingegebene Variablen berücksichtigt werden.
Wir implementieren unser Projekt hauptsächlich mithilfe von Assembler-Code und von C-Code.

2 Problemstellung und Spezifikation

Wir bekommen fünf Parameter vom Benutzer, nämlich ***r_start***, ***r_end***, ***i_start***, ***i_end***, ***res*** und ***img***. Auf der reellen Achse wird durch ***r_start*** und ***r_end*** und auf der imaginären Achse durch ***i_start*** sowie ***i_end*** ein bestimmter Bereich festgelegt. Der Parameter ***res*** ist die Auflösung(Schrittweite pro Bildpixel) und ***img*** ist ein Zeiger auf einen Speicherbereich.
Um die Aufgabe zu erfüllen, müssen wir alle mögliche ***ab***-Kombinationen nach Auflösung und den angegebennen Formeln berechnen, indem wir sie per Interationen mit bestimmter Zahl durchgeführen. Wenn die letzte komplexe Zahl nach ***n*** mal Iterationen noch beschränkt ist, fäben wir den entsprechende Pixel schwarz, ansonst weiß.

$$\mathrm{z_{i+1}} = \bar{\mathrm{z}}_\mathrm{i}^2 + \mathrm{c}(\mathrm{i} \geq 0) \quad \mathrm{z}_0 = 0 \tag{1}$$

$$\mathrm{c} = \mathrm{a} + \mathrm{bi} \quad \mathrm{a} \in [-2; 1] \text{ und } \mathrm{b} \in [-1; 1] \tag{2}$$

Wir berechnen Mächtigkeit vom Wertbereich aus ***a*** sowie ***b*** nach den folgenden Formeln. Im folgenden Text nennen wir Mächtigkeit vom Wertbereich aus ***a*** als Anzahl von ***a*** analog Anzahl von ***b***.

$$\mathrm{AnzahlVonA} = \lfloor 3/\mathrm{res} \rfloor + 1 \quad \mathrm{AnzahlVonB} = \lfloor 2/\mathrm{res} \rfloor + 1 \tag{3}$$

a_{i+1} ist Realteil und b_{i+1} ist Imaginaerteil von z_{i+1}. Und (a_{i+1}, b_{i+1}) ist gemäß der Koordinaten in der komplexen Ebene. Wir berechnen a_{i+1} und b_{i+1} durch folgenden Formeln.

$$\mathrm{a_{i+1}} = (\mathrm{a_i^2} - \mathrm{b_i^2}) + \mathrm{a} \quad \mathrm{b_{i+1}} = -2 * \mathrm{a_i} * \mathrm{b_i} + \mathrm{b} \tag{4}$$

Wir haben bemerkt, dass das Problem des Fließkommaüberlaufs nach einer großen Anzahl von Iterationen auftritt. Deshalb haben wir unsere Formel wie folgt optimiert,

um das Problem des Fließkommaüberlaufs bei der Berechung vom Zwischenergebnis nämlich a^2 und b^2 zu vermeiden.

$$\mathrm{a_{i+1}} = (\mathrm{a_i} - \mathrm{b_i}) \cdot (\mathrm{a_i} + \mathrm{b_i}) + \mathrm{a} \tag{5}$$

Wenn (a_{i+1},b_{i+1}) als Koordinaten nach Iteration noch innerhalb der Grenzen des erlaubten Bereichs bleibt, notieren wir dann entsprechend die ***ab***-Gruppen.
In der obigen Bearbeitung müssen wir die Anzahl der Iterationen dokumentieren sowie wie kann man die entsprechenden ***ab***-Gruppen in ***img*** einspeichern. Zeitgleich sind auch die Probleme des Fließkommaüberlaufs und des Rundungsfehler zu lösen. Dies werden wir in dem nächsten Absatz bearbeiten.

3 Lösungsansatz

3.1 BMP

Um das Fraktal zu visualisieren, müssen wir die Kenntnisse in Bezug auf BMP lernen. BMP ist Abkürzung von Windows Bitmap und bezeichnet ein zweidimensionales Rastergrafikformat.[3] Und die Rastergrafiken bestehen aus einer rasterförmigen Anordnung von so genannten Pixeln, denen jeweils eine Farbe zugeordnet ist.[2] BMP-Dateien bestehen aus drei Teilen: dem Dateikopf, dem Informationsblock und den Bilddaten. Der Wert bfSize ist im Dateikopf und die Größe der BMP-Datei in Byte.

$$\mathrm{bfSize} = 54 + ((\mathrm{biWidth} + \mathrm{AnzahlVonPaddingZero}) * 3) * \mathrm{biHeight} \tag{6}$$

54 ist gleich die Größe von Bitmapfileheader plus die Größe von Bitmapinfoheader. Die Bitmapinfoheader-Struktur dient der Beschreibung der Bitmap. Mit den Werten biWidth und biHeight geben wir die Breite bzw. Höhe der Bitmap in Pixeln an.

$$\mathrm{biWidht} = \mathrm{AnzahlVonA} \tag{7}$$

$$\mathrm{biHeight} = \mathrm{AnzahlVonB} \tag{8}$$

Die Bilddaten werden Zeile für Zeile gespeichert. Wir müssen jede Bildzeile durch rechtsseitiges Auffüllen mit Nullen auf ein ganzzahliges Vielfaches von 4 Bytes ausrichten.
Dann schauen wir uns die Bilddaten an. Wir benutzen 24 bpp,[3] d.h. die Daten jedes Pixels bestehen aus jeweils einem Byte für den Blau-, Grün- und Rot-Kanal. Deshalb drücken #ffffff weiß und #000000 schwarz aus.

3.2 Daten speichern

In dem C-Code reservieren wir zuerst genügenden Platz für Assembler-Code und dann gilt es die Adresse als Parameter zusammen mit den anderen vier vom Benutzer eingegebenen Parametern an den Assembler-Code weiterzuleiten.
In dem Assembler-Code müssen wir jede ***ab***-Gruppe beurteilen. Nach dem Vergleich

des Iterationsergebnisses mit den obigenannten Kreterien müssen wir eine Konstante #000000 oder #ffffff per Bearbeitung in Assembler-Code in die Adresse schreiben. Schließlich soll der Inhalt in der Adresse zuerst per C-Code gelesen werden. Und dann muss der Inhalt für jeder Zeile durch 0 erüllt werden. Und zum Schluss kann ein Bild generiert werden. Und dann schreiben wir die BMP-Datei in die vom Benutzer vorgegebenen Ausgabedatei.

3.3 Fließkommazahl

3.3.1 Rundungsfehler

Aus der Eigenschaft der Fließkommazahl ergibt sich, dass Fließkommazahl Rundungsfehler enthalten. Um BMP-Datei richtig zu erzeugen, müssen wir sicherstellen, dass jede Zeile die gleiche Anzahl von ***ab***-Gruppe aufweist. Deshalb können wir keine While-Schleife verwenden, indem wir jeweils res addieren, um festzustellen, ob die Anzahl der Schleifen den Wertebereich von ***a*** oder ***b*** überschreitet. Einerseits ist, dass der Wert von res nach jeder Iteration einen Fehler enthält und die Fehlers sich auch häufen. Und dann kann dies dazu führen, dass die Schleifenanzahl falsch ist. Andererseits ist Ungenauigkeit der Fließkommazahl. D.h. wenn die Zahl sich in der Nähe der Grenze befindet, kann nicht genau bestimmt werden, ob die Zahl größer als die Grenze ist.
Stattdessen verwenden wir for-Schleife, um die Anzahl der Schleifen auf der Basis der zuvor kalkulierten Anzahl von a und b zu bestimmen. Wegen der Existenz von Rundungsfehlern können wir die Anzahl von a und b nicht unmittelbar durch die obige Formel berechnen, sondern wir müssen $\lfloor x \rfloor$ mit C-Code durch die folgende Formel berechnen

$$\text{Sei } e = \lceil x \rceil - x, x \Leftarrow \begin{cases} \text{round}(x) + 1, \text{ wenn } e < 0.01 \\ \lfloor x \rfloor + 1, \text{ sonst} \end{cases} \tag{9}$$

und mit Assembler-Code durch die folgende Formel berechnen.

$$\begin{aligned} &\text{Sei } y = x + 0.5 \\ &e = \lceil y \rceil - y, x \Leftarrow \begin{cases} \text{round } (y + 0.5) - 1, \text{ wenn } |e - 0.5| < 0.01 \\ \text{round } (y) - 1, \text{ sonst} \end{cases} \end{aligned} \tag{10}$$

3.3.2 Besonderer Wert

Bei der Berechnung sind einige spezielle Werte entstanden, die nicht direkt durch normalen Vergleich beurteilt werden können, ob sie größer als die Grenze sind, hier wie z.B. INF und NAN. Deshalb müssen wir uns mit diesen speziellen Werten separat auseinandersetzen.
Wir bestimmen eine Fließkommazahl direkt durch ucomiss, ob sie NaN(Not a Number) ist. Anschließend wird der absolute Wert dieser Zahl genommen und mit *FLT_MAX* verglichen, um zu bestimmen, ob die Zahl inf ist, denn Inf(Infinity) ist größer als alle Zahlen.

3.4 Anzahl der Iterationen

Um die Anzahl von Iterationen am besten zu entscheiden, haben wir einen Test in der Programmiersprache C geschrieben, indem wir so viel Situationen wie möglich laufen und eine mindestens benötigende Iterationsanzahl zusammenfassen. Denn nach Eigenschaft der Mandelbrot-Menge wissen wir: für alle c gehört zu Mandelbrot-Menge, soll $|z_n| \leq 2$, legen wir die erste vier Parameter in Funktion 'multicorn' fest, Wir nehmen hierbei noch an, dass die Auflösung mit 0.001 an fängt. Wir erhöhen die Anzahl der Iterationen kontinuierlich und gleichzeitig bilden wir das Bild aus jeder Ausführung mit dieser neusten Zahl als Eingabe. Dieses Bild vergleichen wir mit dem bei 500 Iterationen gebildeten Bild. Sobald beide Bilder eine mehr als 99.9%ige Ähnlichkeit aufweisen, notieren wir die momentane Iterationsanzahl un prüfen wir die nächsten fünf Bilder aus den weiteren Iterationen. Wenn alle fünf die Bedingung erüllen können, halten wir diese Anzahl von Iterationen für die beste Anzahl von Iterationen. Nachdem wir eine gasamte Statistik aller Auflösungen, die im Intervall [0.001;1] mit einem Abstand von 0.001 zu erstellen sind, auf die gleiche Weise getestet haben, haben wir folgenden Daten bekommen.

```
Arguments:
r_start:-2.000000, r_end:2.000000, i_start:-2.000000, i_end:2.000000
res: from 0.001000 to 1, step:0.001000, total:1000
  res:0.001000 => min_Iteration_num:219
  res:0.002000 => min_Iteration_num:216
  res:0.003000 => min_Iteration_num:220
             ............
  res:0.998000 => min_Iteration_num:5
  res:0.999000 => min_Iteration_num:6
  res:1.000000 => min_Iteration_num:2
```

Um eine angemessene Iterationsanzahl für allgemeine Situation davon zu finden, benutzen wir die Funktion TRIMMEAN(array, 0.1) von Excel. TRIMMEAN schließt zuerst 5% Werte vom oberen und 5% von unteren Rand des Datensatzes aus und berechnet dann den Durchschnitt. Das Ergebnis ist 69.1544. Und bei anderen Testen mit verschiedenen Grenzen und Auflösung ist es analog. Deshalb benutzen wir danach 70 als die Anzahl von Iterationen.

4 Dokumentation der Implementierung

4.1 Benutzer-Dokumentation

* ***USAGE:tricorn [options]***

* ***-a, –r_start float***
  -Legen Sie einen Gleitkommawert als Untergrenze der reellen Zahl fest. Er sollte nicht grösser als ***r_end*** sein.

* ***-b, –r_end float***

-Geben Sie einen Gleitkommawert als Obergrenze der reellen Zahl ein. Er sollte nicht kleiner als ***r_start*** sein.

* ***-c, –i_start float***
  -Setzen Sie einen Gleitkommawert als Untergrenze der imaginären Zahl. Er sollte nicht grösser als ***i_end*** sein.

* ***-d, –i_end float***
  -Setzen Sie einen Gleitkommawert als Obergrenze der imaginären Zahl. Er sollte nicht kleiner als ***i_start*** sein.

* ***-r, –res float***
  -Legen Sie einen Gleitkommawert als Auflösung für Pixel fest

* ***-o, –output location***
  - Geben Sie eine Adresse ein, die genügend Speicherplätze haben muss.

* ***-h, –help***
  -Diese Nachricht drucken und beenden

Der empfohlene Wertebereich von ***r_start***, ***r_end***, ***i_start***, ***i_end***, mit dem man das ganze Bild bekommen kann, lautet $[2, float\_max]$. Und ***res*** muss größer als 0 sein, am besten größer als 0,00001 sein. Denn wenn res zu klein ist, ist die BMP-Datei zu groß. Das Programm überprüft nicht nur Anzahl von Parametern, sondern auch die Korrektheit der Eingaben. Wenn ungültige Parameter eingegeben werden, meldet das Programm Fehler, z.B. wenn man nur ***r_start*** eingibt, gibt Programm wie folgend aus:

```
r_end - wrong or not given
usage: ./tricorn [options]
options:
-a, --r_start    [float] -the value as lower bound of real number.
-b, --r_end      [float] -the value as upper bound of real number.
-c, --i_start    [float] -the value as lower bound of imaginary number.
-d, --i_end      [float] -the value as upper bound of imaginary number.
-r, --res        [float] -the value as resolution of the picture.
-o, --output     [string]-the path of the output image file.
-h, --help               -print the help message.
example:
tricorn --r_start -2 --r_end 2 --i_start -2 --i_end 2 --res 0.01 --output
    './output.bmp'
```

4.2 Makefile

Zur Anwendung von Makefile gibt es 16 Befehle: ***all***, ***example***, ***testIterationNum***, ***testCorrection***, ***testPerformance***, ***build***, ***clean***, ***runMain***, ***runMainExample***, ***buildWithCImp***, ***buildTestIterationNum***, ***runTestIterationNum***, ***buildTestCorrection***, ***runTestCorrection***, ***buildTestPerformance***, ***runTestPerformance***. Die ersten 5 dabei sind abhängig von anderen übrigen und werden öfter benutzt, deshalb wird hier diesen Teil mehr fokussiert.

* ***all: build runMain***
  -Die Hauptdatei von Compiler generieren und dann ausführen.
* ***example: build runMainExample***
  -Die Hauptdatei von Compiler generieren und dann mit vordefinierte Eingaben ausführen.
* ***testIterationNum: buildTestIterationNum runTestIterationNum***
  –Datei,die die rationale Nummer für Iteration in Hauptprogram testet, von Compiler generieren und dann ausführen.
* ***testCorrection: buildTestCorrection runTestCorrection***
  –Datei, die die Korrektheit dieses Programs per Vergleich mit dem Standardscode testet, von Compiler generieren und dann ausführen.
* ***testPerformance: buildTestPerformance runTestPerformance***
  – Datei,die die Performance dieses Programs per Vergleich mit dem Standardscode testet, von Compiler generieren und dann ausführen.
* ***clean***
  -Löschen alle frühzeitige Operationen.
* ***buildWithCImp***
  -Kompilieren und führen das ganzen Projekt mit einer C-Datei als Ersetzung des Assemble-Anteils aus.

4.3 Entwickler-Dokumentation

Es gibt insgesamt 4 Programme, nämlich test_performance.c, test_correction.c, iteration_number.c und tricorn.c.

4.3.1 Hauptprogram

Das ganze Program bestehen aus 2 Daten, nämlich tricorn.c und multicorn.S. Das c-Datei bekommt die Argumente vom Benutzer und prüfen die Korrektheit. Wenn alle Argumente richtig sind, läuft das Program wie folgt:
C-Program reserviert den passenden Speicherplatz für img, gibt den Zeiger zusammen mit von Benutzer gegenbenen Argumenten in das Assembler-Program ein. Die Assembler-Datei bearbeitet die Pixels Zeile für Zeile mit SIMD Instructions, d.h. jedes Mal 4 Daten mit dem Formel (4) berechnen, Dummy-Datei wird gefüllt, sofern die Anzahl von Daten weniger als 4 ist. Nach der Berechnung wird die Gültigkeit von den Daten zuerst geprüft, danach wird die Grenze geprüft.

4.3.2 Test-program

Das Test-Program bestehen aus 3 Teile: testIterationNum, testCorrection, testPerformance.

testIterationNum: Das ganze Program bestehen aus 2 Daten, nämlich iteration_number.c und multicorn_c_test_double_with_step.c. Die Hauptidee ist gleich wie (3.4). Die multicorn Function hier wird geändert.Das heißt, wir lassen es Schritt für Schritt rechnen und prüfen, damit wir eine passendere Anzahl der Iteration bestimmen und das ganze Program auch schneller funktioniert.
testCorrection: Das ganze Program bestehen aus 3 Daten, nämlich test_correction.c, multicorn_c_test_double.c und multicorn.S. Die Hauptidee ist gleich wie (5). Indem wir die falsche Pixels im Vergleich zum Standardbild in Rot festsetzen. Und die visualisierte Datei nach dem Lauf des Testes wird im Ordner test_correction gespeichert.
testPerformance: Das ganze Program bestehen aus 3 Daten, nämlich test_performance.c, multicorn_c_test_double.c und multicorn.S. Die Hauptidee ist gleich wie (6).

5 Genauigkeit

Wir müssen jetzt die Genauigkeit testen. Zuerst erzeugen wir Standardbild durch C-Code und der Gedankengang ist wie folgende Pseudocode.

```
For each pixel (a0, b0), do:
{
    init(a, b);
    Loop n times:
        (a, b) <= ( a^2 + b^2 + a0, -2*a*b + b0);
    check_print(a0, b0, a, b);
}
```

Die Genauigkeitsrate entspricht der Anzahl der Pixels, die gleich wie was in Standardbild ist, durch die Anzahl aller Pixel in einen Foto, was durch unseren Program dagestellt ist. Damit die genauer sein kann, berechnen wir einen Durchschnitt zahlreicher Genauigkeiten aus verschiedenen Aufrufen. Die Strategie ist so: wir beginnen mit ***r_start*** = -2, ***r_end*** = 2, ***i_start*** = -2, ***i_end*** = 2. Jedes Mal setzen wir ***r_start*** = ***r_start*** * 2, ***r_end*** = ***r_end*** * 2, ***i_start*** = **i_start** * 2, ***i_end*** = ***i_end*** * 2, bis ***r_start*** und ***i_start*** gleich -2^{99} und ***r_end*** und ***i_end*** gleich 2^{99} sind. Wir müssen Randfall testen. Deshalb testen wir auch den Fall, wenn ***r_start*** und ***i_start*** gleich *FLT_MAX* und ***r_end*** und ***i_end*** gleich *FLT_MAX* sind. Für jede Kombination von Grenzwerten testen wir ***res*** separat. ***res*** ist von 0,001 bis 1 und werden jedes Mal um 0,001 erhöht. Dann zeichnen wir die Genauigkeitsrate dazu auf. Schließlich berechnen wir die durchschnittliche Genauigkeitsrate, die dem Grenzwert entspricht.

```
Arguments:
r_start:-2, r_end:2, i_start:-2, i_end:2
res: from 0.001000 to 1, step:0.001000, total:1000
  res:0.001000 => accuracy:0.998964, diff_pixels:6221
  res:0.002000 => accuracy:0.999304, diff_pixels:1045
                  ..................
    res:1.000000 => accuracy:0.916667, diff_pixels:1
summary: 966/1000 test, 96.600% perfect match,average_accuracy:99.985%
```

Dann berechnen wir die durchschnittliche Genauigkeit basierend auf der gesamten Genauigkeit. Zum Schluss beträgt die endgültige Genauigkeit unseres Programms 99,998%.

```
r_start:−2², r_end:2², i_start:−2², i_end:2²
res: from 0.001000 to 1, step:0.001000, total:1000
summary: 968/1000 test, 96.800% perfect match, average_accuracy:99.997%
                        ....................
r_start:-FLT_MAX, r_end:FLT_MAX,i_start:-FLT_MAX,i_end:FLT_MAX,
res: from 0.001000 to 1, step:0.001000, total:1000
summary: 964/1000 test, 96.400% perfect match, average_accuracy:99.997%

total_summary: total_average_accuracy:99.998%
```

Das folgende Bild ist ein Beispiel über Korrektheit mit der bestimmte Grenze. Wir zeigen nur fehlerhafte Daten im folgenden Diagramm.

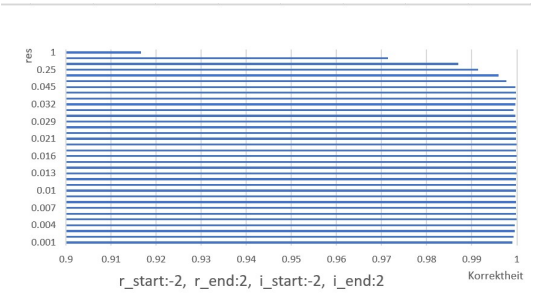

Abbildung 1: r_start=-10,r_rnd=10,i_start=-10,i_end=10

Zum Beispiel wenn der Benutzer die folgenden Parameter eingibt, zeigt unser Programm das folgende Bild.

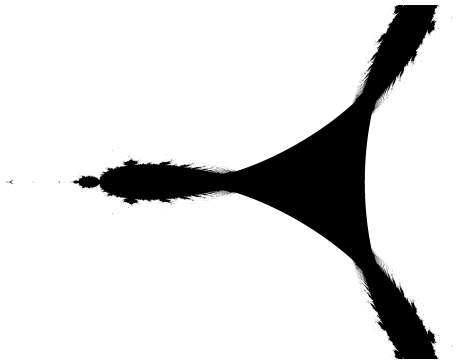

Abbildung 2: r_start=-10,r_rnd=10,i_start=-10,i_end=10,res=0.001

Wir wählen ein exponentielles Wachstum für den Grenzwert anstelle eines linearen Wachstums. Der Grund dafür ist, dass Benutzer eher kleinere Zahlen eingeben. Wir müssen am Anfang mehr Grenzwerte testen. Bei großen Zahlen ist die Wahrschein-

lichkeit einer Benutzereingabe nicht hoch. Wir dürfen weniger Grenzwerte testen. Dies verbessert die Effizienz bei garantierter Genauigkeit.

Unser Programm kann keine 100%ige Genauigkeit erreichen. Es wird einige ungenaue Daten geben. Diese Daten können visualisiert werden. Wie in dem folgenden Bild gezeigt, ist der rote Punkt die Fehlerstelle. Wir glauben jedoch, dass diese Fehler in einem vernachlässigbaren Bereich liegen.

Abbildung 3: Genauigkeit

6 Performanzanalyse

6.1 Performanz

Wir setzen eine Regel: ***res*** erhöht sich von 0,001 auf 1 und jedes Mal um 0,001. Wir berechnen: Die Laufzeit der Program in Assembler und die Laufzeit der entprechenden Program in C. Wir können die folgenden Daten erhalten.

```
 r_start:-2.000000, r_end:2.000000, i_start:-2.000000, i_end:2.000000
res: from 0.001000 to 1, step:0.001000, total:1000
  res:0.001000
    assembly_code:  run_time:0.343072 s
    c_code:   run_time:0.993629 s
  res:0.002000
    assembly_code:  run_time:0.076925 s
    c_code:   run_time:0.248449 s
                ..............
    res:0.999000
    assembly_code:  run_time:0.000001 s
    c_code:   run_time:0.000002 s
  res:1.000000
    assembly_code:  run_time:0.000001 s
    c_code:   run_time:0.000002 s
```

Um deutlicher zu sehen,haben wir auch fogenden Diagramm erstellt.

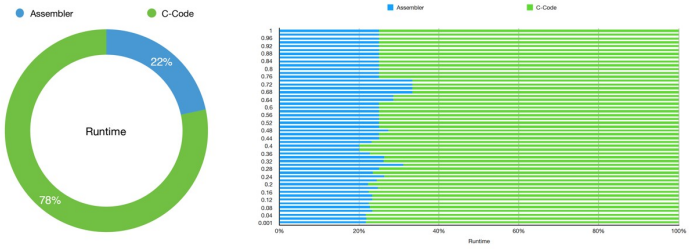

Abbildung 4: Runtime

Durch dieses Diagramm über Laufzeit kann man klar entdecken, dass der Assembler schneller laufen kann. Grund dafür kann man in diene Teil deutlich zeigen: Die folgende beide Code berechnen $\mathrm{a}^2 - \mathrm{b}^2$. Eine ist Assembler-Code und die andere ist Disassembly-Code.

```
1  //Assembler-Code
2  subps   xmm10, xmm14
3  addps   xmm14, xmm13
4  mulps   xmm10, xmm14
5
6  //Disassembly-Code
7  addss   xmm1, DWORD PTR [rbp-0x34]
8  movss   xmm0, DWORD PTR [rbp-0x38]
9  subss   xmm0, DWORD PTR [rbp-0x34]
10 mulss   xmm0, xmm1
```

Im Vergleich zu Disassembly-Code können wir feststellen, dass c während Iteration einmal jeweils nur eine Fleißkommazahl verarbeitet. In unserem Programm verwenden wir während Iteration so viel wie möglich die SIMD-Einheit, um 4 Fließkommazahlen paralle zu verarbeiten.
Geschwindigkeit hängt von zwei Faktoren ab. Eine ist, je größer die Anzahl der Iterationen ist, desto größer ist die Differenz zwischen der Geschwindigkeit von Assembler und der Geschwindigkeit von c. Andere ist, wenn $\mathrm{AnzahlVonA}$ modulo 4 gleich 0 ist, ist schnellste Geschwindigkeit. Wenn $\{\mathrm{AnzahlVonA} \bmod 4 \neq 0\}$, je größer das Ergebnis von $\{\mathrm{AnzahlVonA} \bmod 4\}$ ist, desto schneller ist die Geschwindigkeit. Das ist Randeffekt.

6.2 Weitere Optimierung

Um unser Programm zu optimieren, studieren wir die Grundeigenschaft von Tricorn-Fraktalen. Wir haben eine sehr nützliche Eigenschaft von Tricorn-Fraktalen gefunden, nämlich dass die ***ab***-Gruppe die Bedingung erfüllt: gdw. $\sqrt{a^2 + b^2}$ kleiner als 2.[1]Es

gibt einen kleinen Unterschied zwischen unsere und die Formel von Wikipedia. Unsere Formel ist $z_{i+1} = \bar{z}_i^2 + c(i \geq 0)$. Die Formel von Wikipedia ist $z_{i+1} = z_i^2 + c(i \geq 0)$. Aber wir können beweisen, dass die Eigenschaft für unsere Formel auch gilt. Mathematische Probleme zu lösen ist nicht das Hauptgewicht unserer Aufgabe. Deshalb werden wir sie hier nicht erläutern.
Um diese Formel zu benutzen müssen wir nicht alle möglichen ***ab***-Kombinationen überprüfen, sondern nur die, die diese Voraussetzung erfüllen. Diese Formel verbessert die Laufgeschwindigkeit theoretisch erheblich.

7 Zusammenfassung und Ausblick

Schließlich haben wir das Problem der Bilderzeugung von Tricorn-Fraktalen erfolgreich gelöst. In dem Prozess haben wir unsere Kenntnisse über Fließkommazahlen bzw. Bitmap-Dateien erweitert. Wir haben die Programmoptimierung abgeschlossen und liefern auch Ideen für die mathematische Optimierung. Wir haben auch bemerkt, dass die Leistung von Assembler besser als C-Code ist.
Bis zu diesem Zeitpunkt ist unser Projekt beendet. Wir sind auf viele Schwierigkeiten gestoßen, doch haben wir die überwunden. Wir sind ganz spannend und zufrieden mit unsereren Ergebnissen.

Literatur

[1] Wikipedia. Mandelbrot-menge — wikipedia, die freie enzyklopädie, 2019. [Online; Stand 17. Januar 2020].

[2] Wikipedia. Rastergrafik — wikipedia, die freie enzyklopädie, 2019. [Online; Stand 25. Januar 2020].

[3] Wikipedia. Windows bitmap — wikipedia, die freie enzyklopädie, 2019. [Online; Stand 25. Januar 2020].

[4] Wikipedia. Fraktal — wikipedia, die freie enzyklopädie, 2020. [Online; Stand 25. Januar 2020].

[5] Wikipedia contributors. Tricorn (mathematics) — Wikipedia, the free encyclopedia, 2019. [Online; accessed 28-January-2020].