LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Praktikum Rechnerarchitektur

Gruppe 167 – Abgabe zu Aufgabe A212

Sommersemester 2019

Nicolas Arteaga Manuel Cardenas Erick Quintanar

Inhaltsverzeichnis

1 Einleitung

Diese Dokumentation wurde im Rahmen des Rechnerarchitektur-Praktikums im Sommersemester 2019 angefertigt. Das Praktikum Rechnerarchitektur ist eine Veranstaltung des Lehrstuhls für Rechnerarchitektur und Parallele Systeme an der Technischen Universität München, welche im Zuge des Informatik B.Sc. Studiums absolviert wird. Unsere Aufgabe war es, einen Multicorn Fractal mittels Assembler zu berechnen und dann in einem Bild darzustellen.

Diese Ausarbeitung ist in 7 Teile unterteilt. Die **Einleitung**, wo Sie sich jetzt befinden, soll einen kleinen und prägnanten Überblick des Praktikums darstellen. Im Abschnitt **Problemstellung und Spezifikation** wird die Aufgabenstellung, die einen Interpretätionsspielraum lässt, näher analysiert und spezifiziert. Die **Lösungsfindung** erklärt wie wir die davor vorgestellte Aufgabe gelöst haben und unsere Überlegungen dahinter werden ausführlich erklärt. Die **Dokumentation der Implementierung** erkärt wie unsere Implementierung funktioniert und im Kapitel **Ergebnisse** wird das Resultat genau beschrieben. Als aller letztes geben wir eine kurze **Zusammenfassung** des Ganzen wieder.

Falls Sie auf Unregelmäßigkeiten treffen oder einen Fehler gefunden haben, melden Sie sich bitte bei uns unter einer der folgenden E-Mail-Adressen:
erick.quintanar@hotmail.com, nicolas.arteaga@tum.de, cardenam@in.tum.de

2 Problemstellung und Spezifikation

2.1 Überblick

In unserer Projektaufgabe mussten wir theoretisches Wissen aus der Mathematik anwenden, um einen Algorithmus in Assembler zu Implementieren, der eine bestimmte Art von Multicorn-Fraktal darstellen soll: das Tricorn-Fraktal. Im Folgenden erklären wir unsere Überlegungen zu der offenen Projektaufgabe und was für Annahmen wir bei der Ausarbeitung treffen mussten.

2.2 Theoretischer Teil

Das Multicorn-Fraktal ist durch folgende Formel beschrieben:

$$z_{i+1} = \overline{z_i}^m + c \; (i > 0) \tag{1}$$

Wobei z_i innerhalb der komplexen Ebene zu betrachten ist. Da wir ein Tricorn-Fraktal implementieren sollten, gilt im Folgenden:

$$z_{i+1} = \overline{z_i}^2 + c \; (i > 0) \tag{2}$$

Man fängt dabei mit $z_0 = 0$ an und c wird aus den Koordinaten der komplexen Ebene gewählt. Das heißt, man variiert a und b in $c = (a + bi)$ für $a \in [a_1; a_2]$ und $b \in [b_1; b_2]$,

wodurch verschiedene z_n für jedes eingesetzte c entstehen. Weiter unten, im praktischen Teil der Aufgabenstellung wird die Wahl von a_1, a_2, b_1 und b_2 genauer erklärt.

- Aus (2) berechnen wir Real- sowie Imaginärteil in z_{i+1} für jedes mögliche c innerhalb unserer durch a_1, a_2, b_1 und b_2 beschränkten Intervalle.
- Aus dem Ergebnis von z_i kann man herausfinden, ob für das gegebene c die Formel beschränkt ist. Dadurch kann man das Pixel, das zu c korrespondiert, entweder schwarz oder färbig (die gewählte Farbe beschreibt, wie unbeschränkt das eingesetzte c ist, d.h. wie viele Iterationen gebraucht werden, um zu merken, dass z_i tatsächlich unbeschränkt ist) darstellen.
- Die Formel lässt sich durch benutzung von SIMD für vier verschiedene Werte von c gleichzeitig berechnen.

2.3 Praktischer Teil

In der Datei mit dem Assemblercode mussten wir die Funktion

```
void multicorn(float r_start, float r_end, float i_start, float i_end, float res, unsigned char* img)
```

implementieren, welche die Ergebnisse für alle $c = a + bi$ im Bereich $[a_1, a_2]$ und $[b_1, b_2]$ mit $a_1 =$ `r_start`, $a_2 =$ `r_end`, $b_1 =$ `i_start` und $b_2 =$ `i_end` berechnen sollte. Das Attribut `res` soll dabei die Resolution des Bildes sein und das Attribut `img` ein Pointer auf die Adresse zur Bitmapdatei, wo die Tricorn Menge gezeichnet werden soll.

Dazu muss man für jedes z_i bestimmen, ob die Formel nach einem beliebigen aber fixen i (Anzahl Iterationen) beschränkt ist oder nicht, und dazu den zugehörigen Pixel schwarz färben. Wenn es nicht beschränkt ist, muss der dazugehörige Pixel farbig gemalt werden. Es muss auch gelten, dass der reelle sowie auch der imaginäre Bereich, gegeben durch den Benutzer mit `r_start, r_end, i_start, i_end`, eingehalten wird. Die Resolution des Bildes, d.h. wie scharf es aussehen soll, wird vom Benutzer durch das Attribut `res` gegeben. Das letzte Attribut `img` soll ein Pointer zum Speicherplatz einer 24-bit BMP Datei sein, dieser muss aber nicht vom Benutzer kommen, sondern anderseits von einer in C geschriebenen Datei, die alle I/O Operationen behandeln muss.

Am Ende muss dann nach Ausführen unseres Programms (laut Wikipedia) ein Bild ähnlich zu Abbildung 1 entstehen, es darf aber natürlich auch färbig aussehen.

3 Lösungsfindung

Mit einem klaren Verständnis der Aufgabe, sind wir nun zur Implementierung gesprungen. Im folgenden Kapitel erklären wir unseren Weg zur Lösung. Dazu zählt, welche

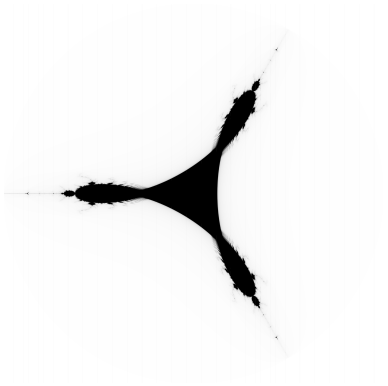


Abbildung 1: Wie ein Tricorn-Fraktal laut Wikipedia aussehen soll [1]

Fragen entstanden sind und wie wir diese gelöst haben, welche Probleme während der Implementierung entstanden und wie wir diese behoben haben. Im Allgemeinen erklären also die nächsten Absätze unsere Entscheidungen und wie diese zu einer funktionalen Implementierung geführt haben.

3.1 Erste Fragen

Nach genauerem Anschauen der Funktion, die wir in Assembler implementieren sollten:

```
void multicorn(float r_start, float r_end, float i_start, float i_end, float res, unsigned
char* img)
```

entstanden erstmal einige Fragen. Zuerst haben wir uns überlegt, wie wir das Attribut `res` betrachten sollten. Bei unserer Implementierung haben wir `res` als ein Skalar des Bildes betrachtet, der die Breite (3) und Länge (2) multipliziert. Als Beispiel: wenn unsere Resolution 100 wäre, wäre am Ende das Ergebnis 300x200 Pixel groß. Das heißt, dass wir aus dem eingesetzten `res` auch die entsprechende Skalierung von jedem Pixel durch $1/res$ berechnen können.

Die zweite Frage, die wir beantworten mussten, ist wie die Berechnung über die Beschränkung von z_i sein soll. Dazu haben wir uns entschieden, dass, wenn nach i Iterationen der Wert von z_i aus dem reellen Bereich $[-2, 1]$ oder aus dem imaginären Bereich $[2, -2]$ herausrutschen sollte, es als unbeschränkt gelten soll und dadurch der Pixel der dem Entsprechenden z_i entspricht (d.h. der Pixel der dem gegebenen c entspricht) musste als Unbeschränkt dargestellt werden.

Auf der anderen Seite ist ein z_i beschränkt, wenn es nach einer fixen Anzahl R von i Iterationen, nicht nach obiger Definition unbescränkt ist. Wir haben R durch einfaches Ausprobieren zu $R = 100$ gesetzt. Es hat sich herausgestellt, dass $R = 50$ die Schärfe des Bildes verringert und dass der Unterschied zwischen $R = 200$ und $R = 100$ zu niedrig

war, damit sich die weiteren Iterationen lohnen würden.

3.2 Technische Entscheidungen

3.2.1 Eingabe

In Bezug auf die Eingabe des Codes gibt, wie oben erklärt, die Projektaufgabe an, dass der Benutzer in der Lage sein soll, die Auflösung des Bildes, sowie die Bereiche komplexer Zahlen, getrennt in Real- und Imaginärteile, zu senden. Diese Eingaben werden dann zur Durchführung der Berechnungen verwendet. Aus diesem Grund haben wir die Auswirkungen der Eingabedaten auf die Berechnungen ausführlich überlegt um unkorrektes Codeverhalten zu vermeiden.

Wir stellen fest, dass die Auflösung nicht Null sein kann, da keine definierte Berechnung stattfinden würde (wir teilen ja eins durch `res`), und dass die Auflösung nicht größer als 5000 sein kann, da die Datei, in der sich das Ergebnis befindet, viel zu viel Speicherplatz (ca. 500 MB) belegen würde. Wir haben uns auch dafür entschieden, dass die Bereiche der imaginären Zahlen sowie der reellen Zahlen nicht Null sein dürfen und dass der Anfang des Bereichs immer kleiner als das Ende sein sollte, da das Programm in beiden Fällen nichts berechnen wuürde. Schließlich überprüft der Code, ob der Benutzer alle erforderlichen Argumente geschrieben hat, um die Berechnung ohne Fehler durchführen zu können.

Es sollte auch beachtet werden, dass die Argumente, die der Benutzer in der Konsole schreibt, als Argumente übergeben werden und dass diese von Strings in Floats umgewandelt werden müssen, damit sie später bei der Berechnung des Fraktals verwendet werden können. Auf diese Weise kann der Benutzer entscheiden, welche Parameter berechnet werden sollen, was dem Benutzer eine große Flexibilität im Bezug auf die Eingabe des Fraktals gibt und somit mögliche Fehler, die sich aus kleinen Versehen ergeben würden, vermeidet. Wir gehen jedoch davon aus, dass der Benutzer die Dokumentation gelesen hat und den korrekten Typen entsprechende Argumente angibt.

3.2.2 Ausgabe

Um das Ergebnis der Berechnungen betrachten zu können, haben wir uns dafür entschieden, eine BMP Datei mit einem Format von 24-Bits pro Pixel zu nutzen. Obwohl eine 32-Bit Datei einfacher zu behandeln ist (da diese kein Padding benötigt), entspricht eine Vergrößerung des Speicherplatzes um acht Bits pro Pixel eine zu hohe und nicht nötige Belastung des Speichers. Auf diesem Grund haben wir die Idee gelassen. Um eine Bitmap von 24 Bits behandeln zu können, ist es ein Muss, dass jede Reihe einen Vielfaches von 32 ist. Daher werden am Ende jeder Reihe die nötigen Bits als Padding eingesetzt.

Die Struktur der Bitmap Datei ist aus [2] entnommen, wo die die einzelnen Headerfelder der Datei erklärt werden. Bevor man die Funktion `multicorn` aufruft, bildet man einen Pointer der zu einem Speicherbereich mit genügend Platz für alle Pixel der Bitmap (inklusive Padding) zeigt. Nachdem `multicorn` die Pixel in dem gegebenen Speicherbereich speichert, errichtet `main.c` eine Datei mit dem davor definierten Header und dem Pointer wo `multicorn` die Werte gespeichert hat. Somit wird das Bild als "result_asm.bmp"gespeichert.

3.2.3 Calling Convention

Damit die Kommunikation zwischen `multicorn` und den Funktionen, die die Werte der Pixel berechnen (`tricorn_asm` und `tricorn_simd`) funktioniert, ohne Registerwerte zu verlieren, haben wir uns auf eine für uns geeignete Calling Convention geeinigt. Falls eine Funktion also ein callee-saved Register benutzt, muss es den alten Wert vorher auf dem Stack speichern und danach wiederherstellen.

Nach unserer definierten Konvention sind die xmm Register 0 bis 5 temporär. Die anderen xmm sind anderseits callee-saved. Bei den 64-Bit Registern gelten Folgende als callee-saved: `RBX`, `RBP`, `RDI`, `RDI`, `RSP`, `R12`, `R13`, `R14` und `R15`. Die anderen sind caller-saved (`RAX`, `RCX`, `RDX`, `R8`, `R9`, `R10`, `R11`). Die übrigen Register werden nicht benutzt und sind dementsprechend für unsere Aufgabe irrelevant.

3.2.4 Assembler Implementierung

3.2.5 Parallelisierung

Unser Hauptziel ist die Berechnung so schnell wie möglich zu machen. Assembler eignet sich besonders gut dazu, da wir einige Berechnungen des Fraktals parallelisieren können. Durch SIMD können wir vier Pixel gleichzeitig (d.h. vier z_i für vier Werte von c) berechnen. Die Speicherung in dem Speicher geschieht aber immer noch Pixel pro Pixel. Das davor erklärte Padding ist hierfür jedoch ein Problem. Wenn vier Nachbarpixels berechnet werden, könnte es geschehen, dass ein Pixel zu einer neuen Reihe gehört. Dementsprechend muss man dazwischen einen Padding in unserem Bild einsetzen.

Als Lösung dazu haben wir uns entschieden, die parallelen Berechnungen Reihe für Reihe durchzuführen. Falls die verbleibenden Pixel weniger als vier sind, berechnen wir diese ohne Parallelisierung und können das eventuell notwendige Padding hinzufügen, womit wir sicherstellen können, das die Länge der Reihen immer ein Vielfaches von 32 Bit ist. Dann können wir mit der nächsten Reihe weitermachen. Somit können die Berechnungen viel schneller ablaufen, da Parallelisierung so oft wie möglich zustande kommt.

4 Dokumentation der Implementierung

Nach einigen Tage Recherche und Analyse hatten wir all unsere Fragen geklärt und wir konnten nun mit der Implementierung anfangen. Zuerst haben wir haben unsere Tricorn-Fraktal in C implementiert und anschließend schrittweise alles zu Assembler umgeschrieben. So konnten wir schrittweise die Funktionen testen und das Debuggen vereinfachen. In den nächsten Zeilen werden wir unsere Lösung und Implementierung genau vorstellen, sowohl aus Sicht eines Benutzers als auch aus der Sicht eines Entwicklers.

4.1 Allgemein

Unsere Implementierung ist in drei Ordner unterteilt: **C**, **Assembly** und **Assembly simd**. Wie man aus den Namen folgern kann, ist in einem Folder die in C geschriebene Implementierung und die Assembler-Implementierungen sind in eine mit Parallelisierung (Assembly simd) und ohne Parallelisierung unterteilt.

4.2 Benutzerdokumentation

In jedem Folder gibt es einen Makefile, das durch Ausführen des Kommandos `make` unsere Implementierung kompiliert. Danach kann man mittels

```
./main r_start r_end i_start i_end res
```

die jeweillige Implementierung ausführen. Die Attribute des Kommandos entsprechen den Attribute der Funktion

```
void multicorn(float r_start, float r_end, float i_start, float i_end, float res, unsigned char* img).
```

Das heißt, dass man einen beliebigen Bereich des tricorns berechnen lassen kann, um es mit beliebiger Resolution anzuschauen. Um eine erste Perspektive zu bekommen, empfehlen wir das Ausführen von:

```
./main -2 -1 -1.5 1.5 1000
```

So kann man das gesamten Tricorn-Fraktal in einer genügend großen Auflösung anschauen (siehe Abb. 2). Das Ergebnis ist in demselben Folder als Bitmap Datei zu finden. Das Spielen mit verschiedenen Attribute ist erwartet und wünschenswert, denn so kann man letztendlich die Schönheit des Tricorn-Fraktals sehen.

Die von `make` generierten Dateien kann man durch Ausführung des Commandos `make clean` schnell löschen.

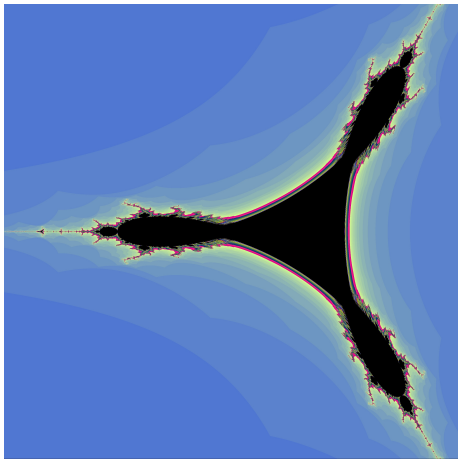


Abbildung 2: Das Ergebnis unserer Implementierung nach Ausführen von `./main -2 -1 -1.5 1.5 1000`

4.3 Entwicklerdokumentation

Unsere Implementierung haben wir in C angefangen. Dann haben wir diese zu Assembler umgewandelt (wobei die ganzen I/O Operationen und die Konfiguration des Bitmaps in C blieben) und zuletzt haben wir die Parallelisierung mit simd implementiert. In den nächsten Absätzen erklären wir die drei Implementierungen.

4.3.1 Erste Implementierung: C

Da Assembler schwer für unsere ersten schnellen Vorstellungen war, haben wir unsere Ideen erstmal in C geschrieben. Wir hoffen, dass nach dieser Erklärung der Leser auch die grobe Funktionsweise des Programms versteht.

Hier haben wir die Funktionalität in zwei Dateien unterteilt. In die Datei, die später in Assembler umgewandelt werden soll (`tricorn.c`) und die Datei, die nach die Erweiterung mit Assembler erhalten bleiben soll (`main.c`), da in dieser die I/O Operationen und die Konfiguration der Bitmap geschehen.

main.c In `main.c` berechnen wir das Padding für das generierte .bmp File und starten die Berechnungen in `tricorn.c` .

tricorn.c In `tricorn.c` berechnen wir die Formel des Tricorns (2) für jedes c innerhalb des eingegebenen Intervalls und für die eingegebene Auflösung.

Die erste Funktion ist `void squareConjugate(float* result)`. Diese bekommt als Parameter ein zweielementiges Array mit dem reellen (`result[0]`) und imaginären (`result[1]`)

Teil der einem gegebenen z_i entsprechen soll. Diese Funktion wandelt gemäß dem Namen also z_i in $\overline{z_i}^2$ um.

Die zweite Funktion ist `void complexAddition(float* result, float* c)`. Diese bekommt die komplexen Zahlen z_i (`result`) und c (`c`) als Parameter und führt $z_i = z_i + c$ aus.

Danach kommt `int check(float* z)`, welche 1 zurückgibt, wenn das gegebene z unstabil ist (d.h. wenn die Funktion aus dem Bereich $[-2, 2]$ sowohl im Imaginären als auch im Reellen ausbricht). Sie gibt 0 zurück, wenn das gegebene z_i (noch) unstabil ist.

Mit `int tricorn(float* z, float* c` wird über ein gegebenes aber fixes z iteriert. In jeder Iteration wird $\overline{z_i}^2$ berechnet und dann c zu z_i addiert. Dies geschieht bis die maximale Anzahl an Iterationen (wir haben uns ja davor für $R = 100$ entschieden) erreicht wird. In jedem Iterationsschritt wird überprüft, ob der Wert noch beschränkt ist. Wenn dies nicht der Fall ist, wird die Anzahl an gebrauchten Iterationen zurückgegeben.

Zuletzt, mit `extern void multicorn_c(float r_start, float r_end, float i_start, float i_end, float res, unsigned char* img)` wird die Bildverarbeitung gemacht. Es wird erstmal die gesamte Anzahl an Bits berechnet, dann genügend Speicher alloziert und anschließend über diesen Speicher (also das Bild) Reihe für Reihe iteriert und die Pixel entsprechend gefärbt. Die Farbe ist abhängig von der Anzahl an Iterationen i.

4.3.2 Assembler

Nachdem wir mit unserer oben beschriebenen Implementierung fertig geworden sind, haben wir `tricorn.c` in Assembler umgewandelt. Dies geschah erstmal schrittweise, Funktion für Funktion. Am Ende sind zwei Dateien entstanden: `tricorn.S`, welches die Formel (2) berechnet und `multicorn.S`, welches die Iterationen bewerkstelligt.

Die größte Veränderung bezüglich C, die durchgeführt wurde, ist, dass wir den Speicher in Assembler nur für die Manipulation des Bildes nutzen. Ansonsten haben wir uns entschieden die verschiedene Werte für z_i und c in Registern zu speichern und weiterzugeben. In den Kommentaren des Codes wird die Benutzung der Register ausführlich erklärt.

4.3.3 Assembler SIMD

Für unsere Implementation der Parallelisierung berechnen wir gleichzeitig 4 Pixel. Nach jeder Berechnung von `SquareConjugate_simd` und `SquareAddition_simd` inkrementieren wir den Counter eines jeden Pixels, dessen z_i noch als nicht unbeschränkt gilt. Die anderen lassen wir unberührt. Damit unsere Programm effizient funktioniert, testen wir mittels `ptest`, ob alle unsere Zähler null sind und springen in diesem Fall zu `end_tricorn_simd` , weil alle Pixel schon als unbeschränkt gelten und ihre verschiedene Farben schon dargestellt werden können.

Ansonsten bleibt die Funktionsweise ähnlich zu der der nicht parallelisierten Assemblervariante. Die Befehle werden zu SIMD Befehlen geändert (zum Beispeil `movss` zu `movups`) und wir arbeiten mit den ganzen 128 Bits der XMM Registern.

5 Ergebnisse

Das Ergebnis an sich wird in der Abb. 2 gezeigt. Es zeigt die Farbskala die gewählt wurde und die Ergebnisse der Funktion 2. Es zeigt alle Pixel, die beschränkt sind, in schwarz. Die Farbe blau steht hierbei für z_i, die sehr schenll unbeschränkt wurden und rot und gelb für solche, die etwas langsamer divergierten.

5.1 Benchmarking unserer Implementierungen

Die Tabelle 5.1 **(Zeile = Implementierungen ; Spalte = Resolution)** zeigt wie sich unsere verschiedenen Implementierungen in der Ausführungszeit **in Sekunden** unterscheiden. Die Implementierung die in C (mit -O3) geschrieben wurde, weist Zeiten vor, die schneller als unser nicht-paralleler Assembler Code sind. Dies liegt an der Optimierungsstufe die gewählt wurde, da diese C Code stark optimiert. Dies zeigt, dass unsere Implementierung in Assembler noch optimiert werden kann.

Die wichtigsten Ergebnisse in der Tabelle werden in der letzten Zeile gezeigt. Diese zeigen, dass unsere parallele Implementierung viel schneller ist, als sogar das Programm, das mit O3 optimiert wurde. Etwas Wichtiges zu beachten ist, dass die Mehrzahl der Pixel des Tricorn-Fraktals divergieren, und man deswegen nur mit einer großen Resolution wirklich den Unterschied sehen kann.

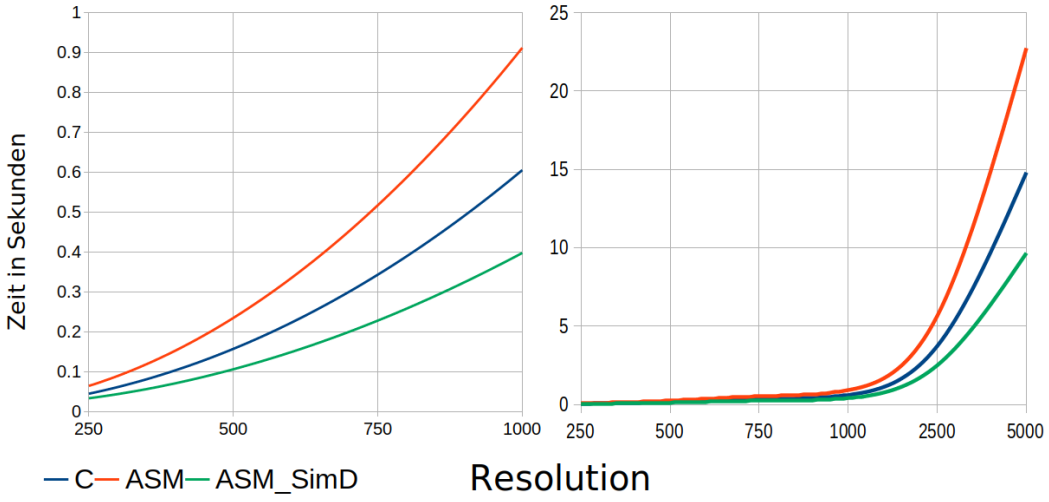
	250	**500**	**750**	**1000**	**2500**	**5000**
C	0.0454124	0.157775	0.3436602	0.605639	3.7216696	14.798096
ASM	0.0650032	0.2352244	0.5176214	0.9119462	5.649717	22.7363514
ASM_SimD	0.03399	0.1066764	0.2285984	0.3981316	2.4923508	9.6546974

Tabelle 1: Prozessor Intel i5-5200U, Zeit für jede Implementierung in Sekunden. [Abb. 3]

6 Zusammenfassung

Zusammengefasst haben wir also eine Aufgabe bekommen, wo wir mittels Assembler einen Tricorn Fractal abhängig von verschiedenen Eingaben des Benutzers berechnen sollen. Nach den ganzen Berechnungen muss also eine Datei wie oben in der Abbildung

Abbildung 3: Visualisierung der Daten aus Tabelle 1[5.1] als Graphen



1 entstehen. Diese darf aber auch gefärbt werden, da die verschiedenen Farben beschreiben, wie unbeschränkt das aktuelle Pixel (also wie unbeschränkt das z_i mit verschiedene c Koordinaten) ist.

Wir haben alles erstmal in C implementiert und dann schrittweise alles in Assembler umgewandelt. Es liest sich aber einfach parallelisieren, denn vier Pixels konnten gleichzeitig berechnet werden. Am Ende haben wir durch ein Benchmark die verschiedene Implementierungen vergliechen und somit als Schlussfolgerung gesehen, dass die Implementierung in SIMD sich doch gelohnt hat, denn die Zeiten viel besser waren.

6.1 Erweiterungsmöglichkeiten

Durch einfaches Betrachten des Bildes merkt man, dass die reelle Achse eine Symmetrieachse darstellt. Das heißt, dass man die Berechnung einer sehr großen Menge an Pixel sparen könnte, um somit die Berechnung noch schneller zu machen. Es ergeben sich auch die Erweiterungsmöglichkeit einer graphischen Benutzeroberfläche, die die Eingabe eventuell noch einfacher gestaltet. Eine weitere naheliegende Idee ist, dem Programm auch die Berechnung von anderer Fraktale wie der Mandelbrotmenge oder der Julia Set beizubringen.

7 Quellenverzeichnis

Literatur

[1] Wikipedia.org: Ein tricorn und seine Merkmale

[2] Wikipedia.org: BMP File Format

[3] Microsoft.com: Device Independent Bitmap

[4] StackOverflow Seiten: Create Simple BMP ; BMP Calculate Size ; Compile BMP with GCC

[5] Rapidtables.com: Hilfe bei die Skalare von die Farben