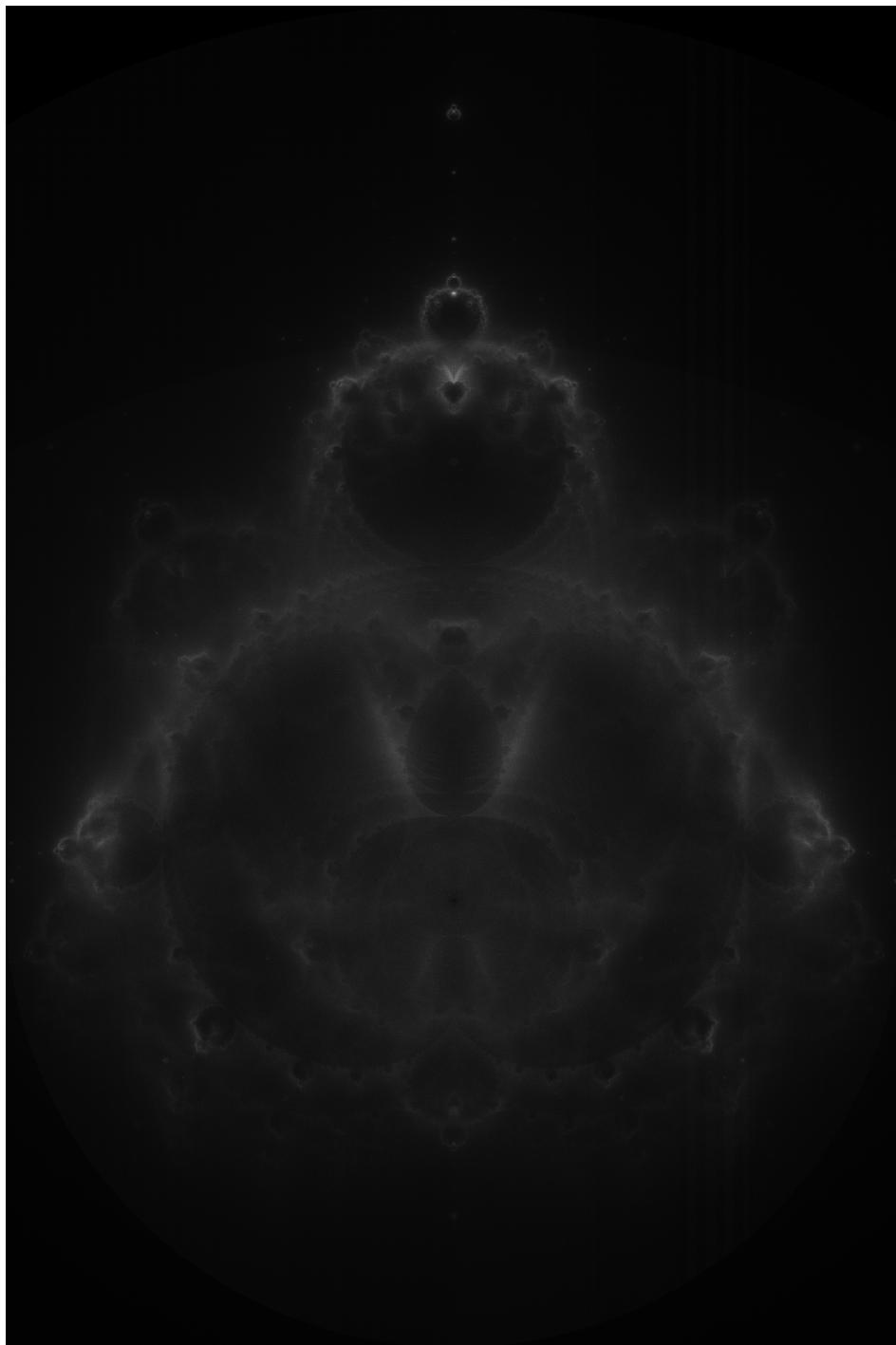


Zoom in das Buddhabrot

Ambros D. Anrig
4. Dezember 2021



Maturaarbeit
Kantonsschule Glarus
Betreuer: Fabio Thöny
Referent: Beat Temperli

Vorwort

Das Unendliche zieht die Menschen schon eh und je an, so auch mich. Es ist unbeschreiblich, doch spielen und rechnen wir damit gerne. Dazu kommt, dass unsere Technik immer besser und leistungsfähiger wird. Dies spielt sich zu, denn desto mehr Leistung man hat, umso schneller und näher kann man sich der Unendlichkeit approximieren. Ich bin schon seit ich klein bin, von komplexeren mathematischen Vorgängen beeindruckt, deshalb schaue ich in meiner Freizeit gerne YouTube-Videos über solche. Eines Tages sah ich das Buddhabrot auf dem Titelbild eines Videofilms. Als ich es mir dann anschauten, verstand ich nichts davon, denn es war auf Englisch. So beschäftigte ich mich vorerst nur mit der Mandelbrot-Menge. Erst als ich in der Schule die Fraktale kennengelernt, verstand ich es besser. So entschloss ich mich nun, das Verstehen des Buddhabrotes nochmals zu versuchen und mich mit dem Buddhabrot auseinander zu setzen. So schreibe ich unter anderem deshalb eine Arbeit darüber.

Ich möchte mich bei einigen Personen bedanken. Zuerst bei meiner Betreuungsperson Fabio Thöny, der mir bei jeglichen Fragen zur Verfügung stand und mich stets unterstützte. Ebenfalls ist meiner Mutter, meiner Schwester und dem Freund meiner Schwester zu danken, die meine Arbeit entgegenlasen und korrigierten. Auch Julian Steiner danke ich, der mir beim Programmieren mental zur Seite stand und mir bei allfälligen Problemen wie auch bei den Formulierungen in der Arbeit geholfen hat. Ebenfalls geht mein Dank an Linus Romer, der mir bei der CUDA-Implementierung half.

Inhaltsverzeichnis

1 Einleitung	4
2 Präliminarien	5
2.1 Komplexe Zahlen	5
2.2 Fraktale Geometrie	5
2.2.1 Allgemein	5
2.2.2 Mandelbrotmenge	6
2.2.3 Buddhabrot	6
3 Methode	8
3.1 Allgemeines Rechnen	8
3.2 Erstellen eines ersten Zooms	8
3.3 CUDA-Optimierung	8
3.4 Analyse	8
3.5 Implementierung der Ergebnisse	9
4 Ergebnisse	10
4.1 Zahlen ohne Optimierung	10
4.2 Analyse der Ergebnisse	10
4.3 Implementierungszahlen	11
5 Diskussion	12
6 Fazit	13
7 Selbständigkeitserklärung	14
8 Literaturverzeichnis	15
9 Anhang	16

1 Einleitung

An der Kantonsschule Glarus werden im Schwerpunkt fach Anwendung der Mathematik und Physik unter anderem auch die Julia-Mengen und die Mandelbrotmenge angeschaut. Kurz wird ebenfalls das Buddhabrot erwähnt. Man schaut auch einen Zoom an, welcher in die Mandelbrotmenge hineinfokussiert. Es sind jedoch wenige Zooms ins Buddhabrot zu finden, geschweige denn solche, welche gleich tief in das Buddhabrot gehen, wie die beim Mandelbrot.

Ziel dieser Arbeit ist es, einen Zoom vom Buddhabrot zu berechnen. Es wird nach einer möglichst effizienten Methode für die Berechnung gesucht. Es soll mit rein mathematischen Algorithmen bewerkstelligt werden, jedoch werden schon zum Anfang leichte Hilfen von Programmiertricks benutzt. Diese Arbeit wird mit der Programmiersprache Julia erstellt, einer schnellen und verständlichen Sprache. Es wird jedoch kein Video erstellt, sondern ein Bild. Ein Video ist eine rasche Abfolge von Bildern. Hat man also eine Methode für das Bild, so ist der Schritt zum Video schon erleichtert. Jedoch würde dieser zusätzliche Schritt den Rahmen dieser Arbeit strapazieren.

2 Präliminarien

2.1 Komplexe Zahlen

Wenn man mit den reellen Zahlen arbeitet, bekommt man Probleme, wenn man die Wurzel aus einer negativen Zahl zieht. Jedoch haben Mathematiker im 17. Jahrhundert eine Lösung für dieses Problem entdeckt, indem dieses Zahlensystem mit den imaginären Zahlen, die die imaginäre Einheit i haben, erweitert wird (Helmuth Gericke 1970, S.66). Addieren und Subtrahieren zweier imaginären Zahlen funktioniert genau gleich, wie wenn man mit einer reellen Zahl rechnet. Dies heisst, dass das i wie die 'herkömmlichen' Variablen behandelt werden kann. Jedoch muss man beim Multiplizieren, Dividieren und beim somit entstehenden Rechnen mit Potenzen aufpassen, denn es gilt für $n \in \mathbb{Z}$:

$$\begin{aligned} i^{4n} &= 1 \\ i^{4n+1} &= i \\ i^{4n+2} &= -1 \\ i^{4n+3} &= -i \end{aligned}$$

Man sollte Potenzen mit der Basis i nach den oben genannten Regeln vereinfachen. Hier sieht man gut, dass eine Verknüpfung zwischen den imaginären und reellen Zahlen in die andere Richtung ebenfalls existiert. Wenn man nun eine imaginäre Zahl ib mit einer reellen Zahl a zusammenaddiert, bekommt man eine komplexe Zahl $c = a + ib$ mit dem Realteil a und dem Imaginärteil b . a und b sind hier reelle Zahlen. Beim Addieren von komplexen Zahlen $z = a + ib$ und $w = e + if$ folgt man diesem Beispiel:

$$\begin{aligned} z + w \\ a + ib + e + if \\ a + e + (b + f)i \end{aligned}$$

Nun merkt man, dass eine komplexe Zahl mit einem Vektor vergleichbar ist, denn um die Zahl darstellen zu können, benutzt man die 2-dimensionale komplexe Ebene (Bertram Maurer 2015, S. 144). Daraus schliesst sich, dass komplexe Zahlen 2-dimensional sind. Multipliziert man eine komplexe Zahl und beobachtet dies auf der komplexen Ebene, fängt der Punkt an scheinbar unkontrolliert herumspringen. Der Punkt folgt jedoch weiterhin logischen Regeln. Beim Quadrieren verschiebt sich der Punkt in die positive Drehrichtung (Gegenuhrzeigersinn). Ebenfalls kann, da die Zahl vergleichbar mit einem Vektor ist, der absolute Betrag der komplexen Zahl c bestimmt werden, welcher mit dem Pythagoras berechnet wird (Reinhart Behr 1989, S. 22):

$$|c| = |a + ib| = \sqrt[2]{a^2 + b^2}$$

2.2 Fraktale Geometrie

2.2.1 Allgemein

Um zum Buddhabrot zu kommen, müssen wir noch einen weiteren Begriff klären: das Fraktal. Fehlt bei einem $3n$ grossem Strich der mittlere Drittel und stehen an dieser Stelle die anderen zwei Seiten eines gleichseitigen Dreiecks mit der Seitenlänge n , so hat man die erste Iteration einer Kochkurve. Fügt man nun in die einzelnen n grosse Striche die vorige Iteration der Kochkurve ein, entsteht die zweite Iteration. Man kann dies ab nun immer wieder machen, sodass ein immer detaillierteres und komplizierteres Bild entsteht.

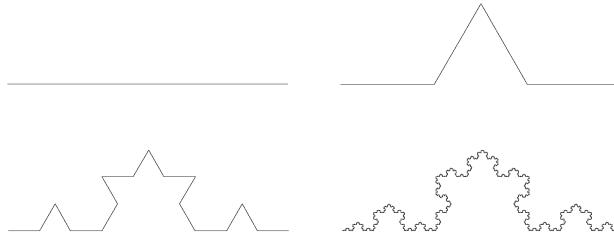


Abbildung 1: Die Entstehung der Kochkurve

Wenn man in die Kochkurve hineinzoomt, findet man die Kochkurve immer wieder: ein rekursives Bild oder eben ein Fraktal (Prof. Dr. Gudio Walz 2001, S. 128).

Man definiert nun das Fraktal als eine Figur, bei der sehr oft Selbstähnlichkeit auffindbar ist, das heisst, dass das gesamte Fraktal oder Teile davon mehrfach im Fraktal vorkommen und das Fraktal selbst eine gebrochene und somit keine ganzzahlige Dimension besitzt (Bertram Maurer 2015, S. 258).

2.2.2 Mandelbrotmenge

Die nach dem Mathematiker Benoît B. Mandelbrot (*20.11.1924; †14.10.2010) benannte Menge (\mathbb{M}) beinhaltet jede Zahl c , die nicht gegen ∞ divergiert für die rekursive Folge (Reinhart Behr 1989, S. 54):

$$\begin{aligned} z_0 &= 0 \\ z_{n+1} &= z_n^2 + c \end{aligned}$$

Man fand heraus, dass wenn $|z_n| > 2$ gilt, wird die Folge gegen ∞ divergieren (Reinhart Behr 1989, S. 74).

Die erstellte Abbildung, je nachdem mit ein bisschen Farbe dazu, ergibt ein sehr schönes Gebilde, welches die deutschsprachigen Leute aufgrund seiner Form an ein 'Apfelmännchen' erinnerte, weshalb es von ihnen auch so genannt wird (Reinhart Behr 1989, S. 54).

Die \mathbb{M} ist ein Fraktal (Bertram Maurer 2015, S. 258). Man findet das erst gesehene Bild der Menge beim Hineinzoomen immer wieder, somit ist es ebenfalls selbstähnlich. Immer wieder findet man verschiedene Julia-Mengen mit dem zugehörigen c (Reinhart Behr 1989, S. 54). Diese sind ebenfalls Fraktale und sehen dazu auch noch für die Norm der Menschheit schön aus. \mathbb{M} besitzt durch die vorgegebene Formel ein chaotisches System (Reinhart Behr 1989, S. 32).

All dies führt sicher dazu, dass es einige YouTube-Videos gibt, die einen Zoom in die Menge zeigen, welche auch viel Rechenleistung brauchen.

2.2.3 Buddhabrot

Schaut man das Wort 'Buddhabrot' als Erstes an, merkt man, dass das Wort 'Buddha' vom meditierenden Buddha kommt, denn dieser ist in der Abbildung ersichtlich. Ebenfalls fällt das Wort 'Brot' auf. Dies ist eine Andeutung, dass diese Abbildung etwas mit dem Mandelbrot zu tun hat, denn es stellt eine andere Variante dar, die \mathbb{M} abzubilden.

Das Bild entsteht, indem das Mandelbrot nochmals berechnet wird, allerdings nur die Punkte, die bei der Mandelbrotberechnung gegen das ∞ divergierten. Nun wird auch nicht mehr geschaut, nach wie vielen Schritten der Punkt c ins ∞ abdriftet, sondern bei welchen Punkten c nach

jeder Iteration landet.

Ebenfalls wird ein Zoom in das Buddhabrot durch das chaotische System von \mathbb{M} und durch die fraktalen und selbstähnlichen Eigenschaften von \mathbb{M} interessant (Melinda Green).

3 Methode

3.1 Allgemeines Rechnen

Um Bilder vom Buddhabrot zu generieren, wurde mit der Programmiersprache Julia eine 2-dimensionale Matrix erstellt, bei der jeder Wert der Matrix zu einem Pixel zugeordnet wird. Zuerst muss man jedoch wissen, welche Punkte man iterieren muss. So wird zuerst das Mandelbrot ausgerechnet. Um Speicherplatz zu sparen, ordnet man ihnen die Werte 0 und 1 zu: 1 divergiert gegen ∞ , 0 gehört der Mandelbrotmenge an. Alle Punkte, die gegen ∞ divergieren, werden nochmals iteriert. Dann wird geschaut, wo die Punkte durchgehen. Am Ende wird geschaut, welcher Punkt die meisten Treffer bekam, denn dieser Wert wird gebraucht, um die Graustufung zu machen. Wie man sicher schon merkt, muss man dreimal die riesige Matrix durchgehen und auch zweimal iterieren. Dies ist somit im Vergleich zur Mandelbrotmenge sehr rechenaufwändig, bei der nur ein Durchlauf nötig wäre. Als Definitionsbereich wird $\{z \in \mathbb{C} \mid -2 < \Re(z) < 1 \& -1 < \Im(z) < 1\}$ gewählt.

3.2 Erstellen eines ersten Zooms

Beim ersten Ansatz wurde für den Zoom eine riesige Matrix erstellt, in die anschliessend mit all den Werten drin, die man braucht, gezoomt wird. Die Grösse der Matrix ist die Zoomtiefe im Quadrat, wie das vom erwarteten Bild. Dann wird ein Ausschnitt gewählt und wird den zeichnen lassen. Dies mithilfe der Berechnung vom Offset im Array, welcher durch die Angabe, zu welchem Punkt man zoomen möchte, berechnet wird.

3.3 CUDA-Optimierung

Um lange Wartezeiten zu vermeiden, wurde der Code mit CUDA formuliert. Es wurde CUDA.jl hinzugefügt, um so mehrere Punkte gleichzeitig rechnen zu lassen. Dadurch, dass nun der grösste Teil des Codes auf der Grafikkarte(GPU) gerechnet wird, ist weniger Speicherplatz verfügbar. Drei Funktionen werden auf der GPU gerechnet: die Berechnung der Punkten die nach ∞ divergieren, die Berechnung des Buddhabrotes und das Zeichnen des Ausschnitts. Für dies müssen die Funktionen umgeschrieben werden, damit CUDA diese kennt. Zu erwähnen ist, dass CUDA eine Plattform ist um auf der Grafikkarte zu rechnen. CUDA wird in Julia durch CUDA.jl genutzt. CUDA ist von Nvidia, was dazu führt, dass das Programm nur noch auf Computern laufen kann, die eine Grafikkarte von Nvidia haben.

3.4 Analyse

Da das Buddhabrot als chaotisches System gilt, wird versucht im Chaos ein Muster zu finden. Eingangs wurde geschaut, ob ein gegebener Bereich einen überwiegenden Einfluss auf einen Bereich, als die anderen Bereichen oder auch markant keinen Einfluss, ausübt. So könnte man in einem Zoom nur noch diesen Bereich anschauen oder vernachlässigen. Dies wurde einfach bewerkstelligt, indem man Bilder erstellte, die zeigten, wie sich Punkte aus diesen Bereichen iterierten. Zuerst wurden die 4 Quadranten als Startbereiche gewählt.

Anschliessend wurde noch zusätzlich die Überlegung gemacht, dass der absolute Wert von c ebenfalls einen Einfluss haben könnte, wie wenn er kleiner als 1 wäre. Dies wurde getestet, indem man eine Variable dem vorigen Aufbau mitgab, welcher mit einem XOR dafür sorgte, dass entweder der Bereich, bei dem $|c| \geq 1$ gilt, oder der andere ausgewertet wird.

3.5 Implementierung der Ergebnisse

Der zu verwerfende Bereich wird schon in der Mandelbrotberechnung verworfen, um Zeit zu sparen. Bei den anderen nützlichen Ergebnissen (vgl. Kapitel Ergebnisse) wird geschaut, in welchen Bereichen auf den Analysebildern es schwarz ist oder nur ein Treffer gezeigt wird. Danach wird diese Fläche zu einer Funktion umgewandelt. Mit dieser wird geschaut, ob der massgebende Eckpunkt des Zoombereichs in der Fläche ist. Falls es so ist, wird deren Quadrant nicht miteinbezogen. So muss man je nachdem nur noch 2 Quadranten für ein Bild berechnen.

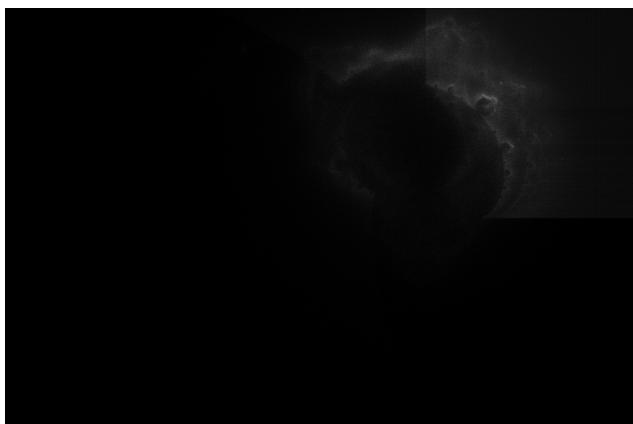
4 Ergebnisse

4.1 Zahlen ohne Optimierung

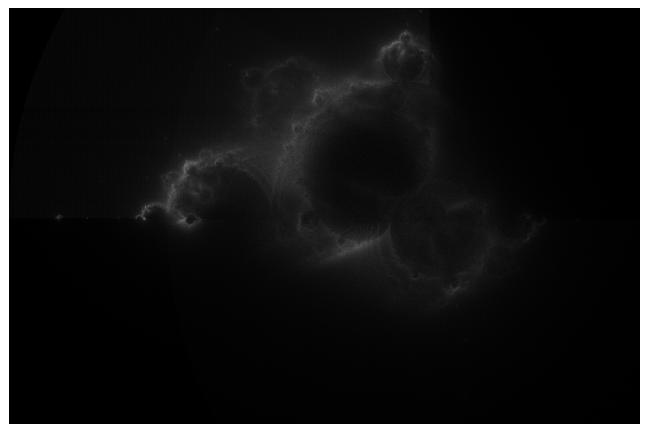
Beim ersten Zoom ist ein 16-facher Zoom nicht mehr möglich, da die Matrix so gross wird, dass der Computer nicht genügend RAM freiräumen kann. Ebenfalls dauert es dann schnell mal viel Zeit, einen 12-fachen Zoom mit der Auflösung 4'001 auf 2'667 Pixel zu berechnen, zum Punkt -1.25 und mit 150 Iterationen etwa 45.6 Stunden.

Durch die CUDA-Optimierung kann man nur noch einen 6.25-fachen Zoom machen, dies liegt daran, dass die GPU nur noch 8GB VRAM hat und der PC 16GB RAM. Bei gleicher Einstellung, bis auf die Zoomtiefe auf 6.25 konnte das Programm innerhalb von 38 Sekunden fertig rechnen. Dass ein Zoom nicht gleich tief möglich ist, ist eigentlich egal, denn es geht ja darum, die Rechenleistung zu verringern und einen allfälligen Algorithmus zu finden. Das gleiche Programm, ein 1-facher Zoom, dauert nun anstelle von 3 Stunden nur noch 2 Minuten.

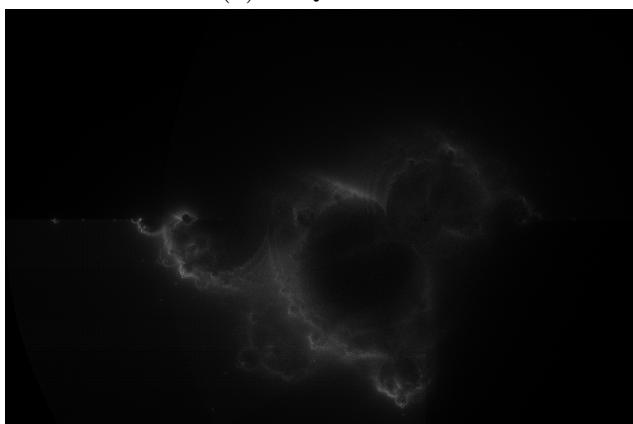
4.2 Analyse der Ergebnisse



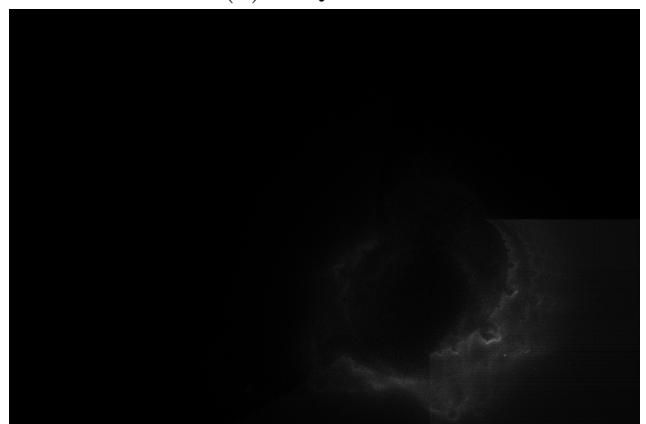
(a) 1. Quadrant



(b) 2. Quadrant



(c) 3. Quadrant



(d) 4. Quadrant

Abbildung 2: Das Buddhabrot der einzelnen Quadranten

Bei Abbildung 3 ist der maximal erreichte Treffer auf einem Pixel vier.

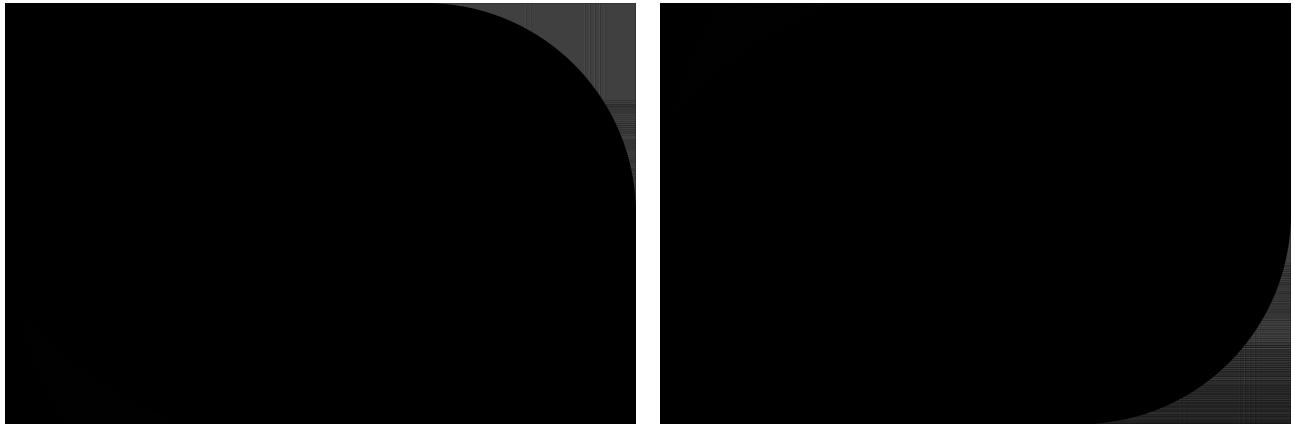


Abbildung 3: Das Buddhabrot für $\{z \in \mathbb{C} \mid |z| > 1\}$ in den Quadranten 1 & 4

4.3 Implementierungszahlen

Durch die gefundene Lösung ist ein 6.48-facher Zoom möglich, jedoch braucht ein gleicher Zoom mehr Zeit. Dies liegt daran, dass nun grosse Matrizen in einer Liste zu finden sind und so der Aufruf durch mehrere if-Kondition länger dauert. Es ist jedoch ein minimaler Verlust und somit nicht schlimm. Ein 6.48-facher Zoom zum Punkt -1.25, mit der Auflösung 4'002 auf 2'668 Pixel und mit 100 Iterationen dauert es nur noch 2 Minuten und 10 Sekunden. Bei einem 6.25-fachen Zoom braucht es 2 Minuten und 1 Sekunde.

5 Diskussion

Bei den Analysen der Quadranten hat sich gezeigt, dass die Quadranten auf bestimmte Bereiche keinen, hingegen auf gewisse Bereiche einen starken Einfluss ausüben. Die Quadranten hatten auf folgende Bereiche im Definitionsbereich keinen Einfluss:

1. Quadrant

$$\{z \in \mathbb{C} \mid \Im(z) > 57.4 \frac{\Re(z)+2}{11.4} - \frac{29.8}{3.8}\}$$

2. Quadrant

$$\{z \in \mathbb{C} \mid ((\Re(z) - \frac{3'504}{667})^2 + \Im(z)^2 > 6.25 \text{ \& } \Im(z) < -\frac{25}{667}) \vee (\Im(z) > (\Re(z) - 1)^2)\}$$

3. Quadrant

$$\{z \in \mathbb{C} \mid ((\Re(z) - \frac{3'504}{667})^2 + \Im(z)^2 > 6.25 \text{ \& } \Im(z) > \frac{25}{667}) \vee (\Im(z) < -(\Re(z) - 1)^2)\}$$

4. Quadrant

$$\{z \in \mathbb{C} \mid \Im(z) < -57.4 \frac{\Re(z)+2}{12} + \frac{24.5}{4}\}$$

Bei den durchgeführten Analysen mit den Radien hat sich gezeigt, dass vom Bereich $\{z \in \mathbb{C} \mid |z| > 1 \text{ \& } 1 \geq \Re(z) \geq 0 \text{ \& } -1 \leq \Im(z) \leq 1\}$ nur ein maximaler Treffer von 4 erreicht wird. Somit kann dieser Bereich in der Berechnung verworfen werden, da 4 in Relation zu den maximal erreichten Treffern von 69 vernachlässigbar ist, da es unter anderem beim Anblick des Buddhabrotes nicht gut erkenntlich ist.

6 Fazit

Eine Steigerung vom ersten Ansatz zur letzten optimierten Fassung ist klar ersichtlich, abgesehen davon, dass das Bild nun ab hellsten ist. Zwar wurde ein Punkt gewählt, bei dem es die letzte Variante gelohnt hat, jedoch sind auch die Punkte in dieser Umgebung sehr spannend. Hätte man einen anderen Punkt ausgewählt, wäre keine klare Verbesserung ersichtlich, wenn denn sogar nicht einmal vorhanden. Ebenfalls ist das letzte Programm nicht gleich effizient wie das zweite, da es nun mehr if-Konditionen hat, welche so das Programm verlangsamen.

Es gibt Einiges, was man hätte probieren können, um einen tieferen Zoom zu machen. Ein Beispiel wäre eine Datenbank, welche während des Rechnens erstellt würde, so dass die Threads bei alten Resultaten hätten weiter rechnen können. Das Programm wäre zwar wiederum langsamer, da nun mehr Aufrufe ausserhalb der CUDA geschehen. Eine andere Methode wäre den Metropolis-Hastings Algorithmus von Alexander Boswell zu nutzen. Dies wurde nicht gemacht, da dort die Wahrscheinlichkeit, dass nicht alle Punkte miteinbezogen werden, vorhanden ist. Hier wird nämlich die Wahrscheinlichkeit vom Divergieren eines Punktes berechnet. Dass alle Punkte miteinbezogen werden, ist durch den Verwerfungsbereich in der letzten Variante dieser Arbeit ebenfalls nicht gegeben, jedoch war der maximale Treffer 4 bei verschiedenen grossen Iterationsstufen. Ebenfalls sind hier die Punkte bestimmt nicht vorhanden und nicht von einer Wahrscheinlichkeit abhängig.

7 Selbständigkeitserklärung

Hiermit bestätige ich, Ambros Daniel Anrig, meine Maturaarbeit selbständig verfasst und alle Quellen angegeben zu haben.

Ich nehme zur Kenntnis, dass meine Arbeit zur Überprüfung der korrekten und vollständigen Angabe der Quellen mit Hilfe einer Software (Plagiaterkennungstool) geprüft wird. Zu meinem eigenen Schutz wird die Software auch dazu verwendet, später eingereichte Arbeiten mit meiner Arbeit elektronisch zu vergleichen und damit Abschriften und eine Verletzung meines Urheberrechts zu verhindern. Falls Verdacht besteht, dass mein Urheberrecht verletzt wurde, erkläre ich mich damit einverstanden, dass die Schulleitung meine Arbeit zu Prüfzwecken herausgibt.

Ort

Datum

Unterschrift

8 Literaturverzeichnis

- [1] Reinhart Behr, *Ein Weg zur fraktalen Geometrie*, Ernst Klett Schulbuchverlag, Stuttgart, 1989.
- [2] Bertram Maurer, *Mathematik - Die faszinierende Welt der Zahlen*, Fackelträger Verlag GmbH, Köln, Emil-Hoffmann-Strasse 1, D-50996 Köln, 2015.
- [3] Prof. Dr. Gudio Walz (ed.), *Lexikon der Mathematik*, Spektrum Akademischer Verlag GmbH Heidelberg, Berlin & Heidelberg, 2001.
- [4] Helmuth Gericke, *Geschichte des Zahlenbegriffs*, Bibliographisches Institut, Mannheim, 1970.
- [5] Melinda Green, *The Buddhabrot Technique* (2017), <https://superliminal.com/fractals/bbrot/>. [Online; Stand 29.11.2021].

9 Anhang

```

1 # Dies ist der erst Versuch der Arbeit
2
3 using Images, Colors
4
5 # variabeln deffinieren
6 @time begin
7     #Varierende variabeln
8     println(@__FILE__)
9     n = Int(2667)
10    n = Int(360)
11    m = Int(floor(n/720*1080))
12    m = 640
13
14    iteration = 1000
15
16    zoom = 1 #zoom != 0
17    zoomPoint = -0.5 + 0im
18
19    #Berechnete variabeln
20    zoomPointAsMatrixPoint = ((-imag(zoomPoint) + 1)*n*zoom/2 + 1, (real(
21        zoomPoint) + 2)*m*zoom/3 + 1)
22    println(zoomPointAsMatrixPoint)
23
24    # verschiebung des Bildes im gesamt array
25    horizontal = Int(floor(zoomPointAsMatrixPoint[1] - n/2))          # zuerst
26    die auf der Komplexenebene rechtere
27    vertical = Int(floor(zoomPointAsMatrixPoint[2] - m/2))          # zuerst die
28    auf der Komplexenebene hoechtere
29    println(string(horizontal) * " " * string(vertical))
30    maxLanding = 0
31
32    # erstellen der array
33    img = zeros(RGB{Float64}, n, m)
34    maxValues = zeros(Int128, n*zoom, m*zoom)
35
36    mandelbrot = zeros(Int8, n*zoom, m*zoom)
37
38    # erstellen der Funktionen
39
40    # bearbeitet das mandelbrot array so das nunroch 1 und 0 gibt, 1 fuer
41    drausen und 0 fuer in der Menge
42    function mandelbrotmenge()
43        for i = 1:(n*zoom)
44            for j = 1:(m*zoom)
45                y = (2*i - n) * 1im # definitionsbereich = [-1im, 1im]
46                x = (3*j - 2*m)/m # definitionsbereich = [-2, 1]
47                z = x + y
48                c = x + y
49                f = []
50                for _ = 1:iteration
51                    if z in f
52                        break
53                    elseif abs(z) >= 2
54                        mandelbrot[i, j] = 1
55                        break
56                    end
57                    append!(f, z)
58                end
59            end
60        end
61    end
62
63    # schreibt das Bild
64    write("mandelbrot.png", img)
65
```

```

54         z = z^2 + c
55     end
56   end
57 end
58
59 # schaut was die Maximale Iterationzahl war, und speichert in maxValues
60 wie oft dort ein Punkt ankam
61 function berechnungBuddhaBrot(i, j)
62   global maxLanding
63
64   y = (2*i - n) / n * 1im # definitionsbereich = [-1im, 1im]
65   x = (3*j - 2*m) / m # definitionsbereich = [-2, 1]
66   z = x + y
67   c = x + y
68   f = []
69   for _ = 1:iteration
70     y = PointImagToIndex(z)
71     x = PointRealToIndex(z)
72
73     if z in f
74       break
75     elseif abs(z) >= 2
76       break
77     elseif true in (x > (m*zoom) , x < 1 , y > (n*zoom) , y < 1)
78       break
79   end
80
81   maxValues[y, x] += 1
82   if maxValues[y, x] > maxLanding
83     maxLanding = maxValues[y, x]
84   end
85   append!(f, z)
86   z = z^2 + c
87 end
88
89 function zeichnen(y, x)
90   return RGB{Float64}(maxValues[y, x]/maxLanding, maxValues[y, x]/
91 maxLanding, maxValues[y, x]/maxLanding)
92 end
93
94 PointImagToIndex(c) = floor(Int64, ((imag(c) + 1) * (n*zoom) / 2))
95
96 PointRealToIndex(c) = floor(Int64, ((real(c) + 2) * (m*zoom) / 3))
97
98 # Hauptprogramm
99 if (real(zoomPoint) - 3/(zoom*2) < -2) || (real(zoomPoint) + 3/(zoom*2)
> 1) || (imag(zoomPoint) - 1/(zoom) < -1) || (imag(zoomPoint) + 1/(zoom)
> 1)
100   println("Zoom auserhalb der Vordefinierte Bildreichweite")
101   exit()
102 end
103
104 mandelbrotmenge()
105
106 for i = 1:(n*zoom)
107   for j = 1:(m*zoom)
108     if mandelbrot[i, j] != 0

```

```

109         berechnungBuddhaBrot(i, j)
110     end
111 end
112 for i = 1:n
113     for j = 1:m
114         img[i, j] = zeichnen(i + horizontal - 1, j + vertical - 1)
115     end
116 end
117
118 # Bildstellung
119 save(string(@__DIR__) * "/Pictures/BuddhabrotmengeWithZoom$(zoom)
120 ToPoint$(zoomPoint)WithIteration$(iteration)withResolution$(m)x$(n).png",
121 img)
122 end

# Dies ist die mit CUDA optimierte Fassung, somit die 2.

using Images, Colors, CUDA

@time begin
    #Varierende variablen
    n = Int(2668)
    m = Int(floor(n/2*3))

    iteration = 100
    anzahlThreads = 256

    zoomPoint = -0.5 + 0im
    zoom = 1

    #Berechnete variablen
    zoomPointAsMatrixPoint = ((-imag(zoomPoint) + 1)*n*zoom/2 + 1, (real(
    zoomPoint) + 2)*m*zoom/3 + 1)
    println(zoomPointAsMatrixPoint)

    # verschiebung des Bildes im gesamt array
    horizontal = Int(floor(zoomPointAsMatrixPoint[1] - n/2))           # zuerst
    die auf der Komplexenebene rechtere
    vertical = Int(floor(zoomPointAsMatrixPoint[2] - m/2))           # zuerst die
    auf der Komplexenebene hoechere
    println(string(horizontal) * " " * string(vertical))

    # erstellen der array
    mandelbrot = CUDA.zeros(Int8, round(Int, n*zoom), round(Int, m*zoom))

    # erstellen der Funktionen

    # bearbeitet das mandelbrot array so das nunroch 1 und 0 gibt, 1 fuer
    drausen und 0 fuer in der Menge
    function mandelbrotmenge!(nzoom::Int64, mzoom::Int64, n::Int64, m::Int64
    , iteration::Int64, mandelbrot, f::CuDeviceVector{ComplexF64, 1})
        indexX = (blockIdx().x - 1) * blockDim().x + threadIdx().x
        strideX = blockDim().x * gridDim().x
        indexY = (blockIdx().y - 1) * blockDim().y + threadIdx().y
        strideY = blockDim().y * gridDim().y

        for i = indexX:strideX:nzoom
            for j = indexY:strideY:mzoom

```

```

39         y = (2*i - n)/n * 1im # definitionbereich = [-1im, 1im]
40         x = (3*j - 2*m)/m # definitionbereich = [-2, 1]
41         z = x + y
42         c = x + y
43
44         for r = 1:iteration
45             if z in f
46                 break
47             elseif abs(z) >= 2
48                 mandelbrot[i, j] += 1
49                 break
50             end
51             f[r] = z
52             z = z^2 + c
53         end
54     end
55     return nothing
56 end
57
58
59 function bench_mandel!(nzoom::Int64, mzoom::Int64, n::Int64, m::Int64,
60 iteration::Int64, mandelbrot)
61     f = CUDA.zeros(ComplexF64, iteration)
62     f .= 3
63     numblocks = ceil(Int, length(mandelbrot)/anzahlThreads)
64     CUDA.@sync begin
65         @cuda threads=anzahlThreads blocks=numblocks mandelbrotmenge!(
66         nzoom, mzoom, n, m, iteration, mandelbrot, f)
67     end
68 end
69
70
71 # schaut was die Maximale Iterationzahl war, und speichert in maxValues
72 wie oft dort ein Punkt ankam
73 function berechnungBuddhaBrot!(nzoom::Int64, mzoom::Int64, n::Int64, m::Int64,
74 iteration::Int64, maxValues, mandelbrot, f::CuDeviceVector{ComplexF64, 1})
75     indexX = (blockIdx().x - 1) * blockDim().x + threadIdx().x
76     strideX = blockDim().x * gridDim().x
77     indexY = (blockIdx().y - 1) * blockDim().y + threadIdx().y
78     strideY = blockDim().y * gridDim().y
79
80     for i = indexX:strideX:nzoom
81         for j = indexY:strideY:mzoom
82             if mandelbrot[i, j] != 0
83                 y = (2*i - n)/n * 1im # definitionbereich = [-1im, 1im]
84                 x = (3*j - 2*m)/m # definitionbereich = [-2, 1]
85                 z = x + y
86                 if !((abs(z) > 1) & (x >= 0))
87                     c = x + y
88                     for r = 1:iteration
89                         y = CUDA.floor(Int64, (imag(z)+1)*nzoom/2)
90                         x = CUDA.floor(Int64, (real(z)+2)*mzoom/3)
91
92                         if z in f
93                             break
94                         elseif abs(z) >= 2
95                             break
96                         end
97                         if (1 < x < mzoom) & (1 < y < nzoom)

```

```

93             maxValues[y, x] += 1
94         end
95
96         f[r] = z
97         z = z^2 + c
98     end
99 end
100 end
101 end
102 end
103 return nothing
104 end

105
106 function bench_buddhi!(nzoom::Int64, mzoom::Int64, n::Int64, m::Int64,
107 iteration::Int64, maxValues, mandelbrot)
108     f = CUDA.zeros(ComplexF64, iteration)
109     f .= 3
110     numblocks = ceil(Int, length(maxValues)/anzahlThreads)
111     CUDA.@sync begin
112         @cuda threads=anzahlThreads blocks=numblocks
113         berechnungBuddhaBrot!(nzoom, mzoom, n, m, iteration, maxValues,
114         mandelbrot, f)
115     end
116 end

117 function zeichnen(n::Int64, m::Int64, horizontal::Int64, vertical::Int64
118 , maxValues, img, maxLanding::Int128)
119     indexX = (blockIdx().x - 1) * blockDim().x + threadIdx().x
120     strideX = blockDim().x * gridDim().x
121     indexY = (blockIdx().y - 1) * blockDim().y + threadIdx().y
122     strideY = blockDim().y * gridDim().y
123
124     for i = indexX:strideX:n
125         for j = indexY:strideY:m
126             img[i, j] = RGB{Float64}(maxValues[i+horizontal-1, j+
127 vertical-1]/maxLanding, maxValues[i+horizontal-1, j+vertical-1]/
128 maxLanding, maxValues[i+horizontal-1, j+vertical-1]/maxLanding)
129         end
130     end
131 end

132 function bench_zeich!(n::Int64, m::Int64, horizontal::Int64, vertical::
133 Int64, maxValues, img, maxLanding::Int128)
134     numblocks = ceil(Int, length(maxValues)/anzahlThreads)
135     CUDA.@sync begin
136         @cuda threads=anzahlThreads blocks=numblocks zeichnen(n, m,
137         horizontal, vertical, maxValues, img, maxLanding)
138     end
139 end

140 # Hauptprogramm
141 if (real(zoomPoint) - 3/(zoom*2) < -2) || (real(zoomPoint) + 3/(zoom*2)
142 > 1) || (imag(zoomPoint) - 1/(zoom) < -1) || (imag(zoomPoint) + 1/(zoom)
143 > 1)
144     println("Zoom auserhalb der Vordefinierte Bildreichweite")
145     exit()
146 end

147 bench_mandel!(round(Int64, n*zoom), round(Int64, m*zoom), n, m,

```

```

iteration, mandelbrot)
maxValues = CUDA.zeros(Int128, round(Int,n*zoom), round(Int,m*zoom))
bench_buddhi!(round(Int64, n*zoom), round(Int64, m*zoom), n, m,
iteration, maxValues, mandelbrot)
mandelbrot = nothing

img = CUDA.zeros(RGB{Float64}, n, m)
println(maximum(maxValues))
bench_zeich!(n, m, horizontal, vertical, maxValues, img, maximum(
maxValues))

# Bildstellung
img_cpu = zeros(RGB{Float64}, n, m)
img_cpu .= img
save(string(@__DIR__) * "/Pictures/gpu/BuddhabrotmengeWithZoomGPU$(zoom)
ToPoint$(zoomPoint)WithIteration$(iteration)withResolution$(m)x$(n)high.
png", img_cpu)
end

```

```

1 # Dies ist die letzte und somit Optimierte Fassung der Arbeit
2
3 using Images, Colors, CUDA
4
5 @time begin
6     #Varierende variabeln
7     n = Int(2668) # muss eine Gerade Zahl sein
8     m = Int(floor(n/2*3))
9
10    iteration = 1000
11    anzahlThreads = 256
12
13    zoomPoint = -.5 + .5im
14    zoom = 6.25 #zoom != 0
15
16    #Berechnete variabeln
17    zoomPointAsMatrixPoint = ((-imag(zoomPoint) + 1)*n*zoom/2 + 1, (real(
18        zoomPoint) + 2)*m*zoom/3 + 1)
19
20    # verschiebung des Bildes im gesamt array
21    horizontal = floor(Int, zoomPointAsMatrixPoint[1] - n/2)           # zuerst
22    die auf der Komplexenebene rechtere
23    horizontal2 = floor(Int, zoomPointAsMatrixPoint[1] + n/2)
24    vertical = floor(Int, zoomPointAsMatrixPoint[2] - m/2)           # zuerst die
25    auf der Komplexenebene hoechtere
26    vertical2 = floor(Int, zoomPointAsMatrixPoint[2] + m/2)
27    zoomPointAsMatrixPoint = nothing
28
29    # erstellen der Funktionen
30
31    # bearbeitet das mandelbrot array so das nunroch 1 und 0 gibt, 1 fuer
32    drausen und 0 fuer in der Menge
33    function mandelbrotberechnung!(nn::Int64, mm::Int64, iteration::Int64,
34    mandelbrotPart, f::CuDeviceVector{ComplexF64}, 1}, r::Int8)
35        indexX = (blockIdx().x - 1) * blockDim().x + threadIdx().x
36        strideX = blockDim().x * gridDim().x
37        indexY = (blockIdx().y - 1) * blockDim().y + threadIdx().y
38        strideY = blockDim().y * gridDim().y
39
40        for i = indexX:strideX:nn
41            for j = indexY:strideY:mm
42                if abs(f[i+j]) >= 4
43                    r[i+j] = 0
44                else
45                    r[i+j] = 1
46                end
47            end
48        end
49    end
50
51    #mandelbrotberechnung!(iteration, n, m, zoom, zoomPointAsMatrixPoint, f, r)
52
53    # Speichern des Bildes
54    save("mandelbrot.png", r)
55
56    # Zeigt das Bild
57    display(r)
58
```

```

36         for j = indexY:strideY:mm
37             y = floor((r-1)/2)*-1 + i/nn # definitionbereich = [-1im, 1
38             im]
39             x = 2*abs(r-2.5)-3 + 2*j/mm # definitionbereich = [-2, 1]
40             z = x + y*1im
41             if !((abs(z) > 1) & (x >= 0))
42                 c = x + y*1im
43
44             for q = 1:iteration
45                 if z in f
46                     break
47                 elseif abs(z) >= 2
48                     mandelbrotPart[i, j] += 1
49                     break
50                 end
51                 f[q] = z
52                 z = z^2 + c
53             end
54         end
55     end
56     return nothing
57 end
58
59 function bench_mandel!(iteration::Int64, mandelbrotPart, r::Int8)
60     f = CUDA.zeros(ComplexF64, iteration)
61     f .= 3
62
63     nn = CUDA.size(mandelbrotPart, 1)
64     mm = CUDA.size(mandelbrotPart, 2)
65
66     numblocks = ceil(Int, length(mandelbrotPart)/anzahlThreads)
67
68     CUDA.@sync begin
69         @cuda threads=anzahlThreads blocks=numblocks
70         mandelbrotberechnung!(nn, mm, iteration, mandelbrotPart, f, r)
71     end
72 end
73
73 # schaut was die Maximale Iterationzahl war, und speichert in maxValues
74 wie oft dort ein Punkt ankam
74 function berechnungBuddhaBrot!(nzoom::Int64, mzoom::Int64, nn::Int64, mm
75 ::Int64, iteration::Int64, maxValues, mandelbrot, f::CuDeviceVector{
75 ComplexF64, 1}, r::Int8)
76     indexY = (blockIdx().x - 1) * blockDim().x + threadIdx().x
77     strideY = blockDim().x * gridDim().x
78     indexX = (blockIdx().y - 1) * blockDim().y + threadIdx().y
79     strideX = blockDim().y * gridDim().y
80
81     for i = indexY:strideY:nn
82         for j = indexX:strideX:mm
83             if mandelbrot[i, j] != 0
84                 y = floor((r-1)/2)*-1 + i/nn # definitionbereich = [-1im
85 , 1im]
86                 x = 2*abs(r-2.5)-3 + 2*j/mm # definitionbereich = [-2,
87 1]
87                 z = x + y*1im
88                 c = x + y*1im
89                 for q = 1:iteration

```

```

88             y = CUDA.floor(Int64, (imag(z)+1)*(nzoom-1)/2+1)
89             x = CUDA.floor(Int64, (real(z)+2)*(mzoom-1)/3+1)
90
91             if z in f # Kontrolle
92                 break
93             elseif abs(z) >= 2
94                 break
95             end
96             if (1 <= x <= mzoom) & (1 <= y <= nzoom)
97                 maxValues[y, x] += 1
98             end
99
100            f[q] = z
101            z = z^2 + c
102        end
103    end
104 end
105
106 return nothing
107 end
108
109 function bench_buddhi!(iteration::Int64, maxValues, mandelbrot, r::Int8)
110     f = CUDA.zeros(ComplexF64, iteration)
111     f .= 3
112
113     nn = CUDA.size(mandelbrot,1)
114     mm = CUDA.size(mandelbrot,2)
115     nzoom = CUDA.size(maxValues,1)
116     mzoom = CUDA.size(maxValues,2)
117
118     numblocks = ceil(Int, length(mandelbrot)/anzahlThreads)
119     CUDA.@sync begin
120         @cuda threads=anzahlThreads blocks=numblocks
121         berechnungBuddhaBrot!(nzoom, mzoom, nn, mm, iteration, maxValues,
122         mandelbrot, f, r)
123     end
124 end
125
126 function zeichnen(n::Int64, m::Int64, horizontal::Int64, vertical::Int64
127 , Values, img, maxLanding::Int128)
128     indexX = (blockIdx().x - 1) * blockDim().x + threadIdx().x
129     strideX = blockDim().x * gridDim().x
130     indexY = (blockIdx().y - 1) * blockDim().y + threadIdx().y
131     strideY = blockDim().y * gridDim().y
132
133     for i = indexX:strideX:n
134         for j = indexY:strideY:m
135             img[i, j] = RGB{Float64}(Values[i+horizontal-1, j+vertical-1]/maxLanding,
136             Values[i+horizontal-1, j+vertical-1]/maxLanding, Values[i+horizontal-1, j+vertical-1]/maxLanding)
137         end
138     end
139 end
140
141 function bench_zeichn!(horizontal::Int64, vertical::Int64, Values, img,
142 maxLanding::Int128)
143     n = CUDA.size(img, 1)
144     m = CUDA.size(img, 2)

```

```

141     numblocks = ceil(Int, length(Values)/anzahlThreads)
142     CUDA.@sync begin
143         @cuda threads=anzahlThreads blocks=numblocks zeichnen(n, m,
144         horizontal, vertical, Values, img, maxLanding)
145     end
146 end
147
148 # mandelbrot
149 mandelbrot = CUDA.Array([CUDA.ones(Int8, 2, 2) for _ in 1:4])
150 maxValues = CUDA.zeros(Int128, floor(Int, n*zoom), floor(Int, m*zoom))
151
152 # Kontrolle ob Zoom vereinfachung moeglich
153 if zoom > 2
154     if (-horizontal2 > (28.7*n)*vertical2/(4*m) - m*9.5/4) #4. Quadrant
155         mandelbrot[1] = zeros(Int8, round(Int, n/2*zoom), round(Int, m*
156         zoom/3))
157     end
158     if (((vertical2-4340)^2+(horizontal2-n/2)^2>(2.5/3*m)^2) &
159         (horizontal2 < n/2-50)) || (horizontal > 9 * n/(2*m^2)*(vertical-m)^2+n/2)
160     ) #3. Quadrant
161         mandelbrot[2] = zeros(Int8, round(Int, n/2*zoom), round(Int, 2*m
162         *zoom/3))
163     end
164     if (((vertical2-4340)^2+(horizontal-n/2)^2>(2.5/3*m)^2) &
165         (horizontal > n/2+50)) || (horizontal2 < -9 * n/(2*m^2)*(vertical-m)^2+n/
166         2)) #2. Quadrant
167         mandelbrot[3] = zeros(Int8, round(Int, n/2*zoom), round(Int, 2*m
168         *zoom/3))
169     end
170     if (horizontal > (28.7*n)*vertical2/(3.8*m) - n*13/3.8) #1. Quadrant
171         mandelbrot[4] = zeros(Int8, round(Int, n/2*zoom), round(Int, m*
172         zoom/3))
173     end
174 else
175     for i = 1:2
176         mandelbrot[i^2] = zeros(Int8, round(Int, n/2*zoom), round(Int, m
177         *zoom/3))
178         mandelbrot[i+1] = zeros(Int8, round(Int, n/2*zoom), round(Int,
179         2*m*zoom/3))
180     end
181 end
182 println("Startet MainProgramm")
183 # Hauptprogramm
184 for r::Int8 = 1:4
185     if CUDA.size(mandelbrot[r]) != CUDA.size(CUDA.zeros(Int8, 2, 2))
186         bench_mandel!(iteration, mandelbrot[r], r)
187         bench_buddhi!(iteration, maxValues, mandelbrot[r], r)
188     end
189 end
190
191 # loeschen und neuerstellen von Arrays aufgrund Speicher-Handling
192 mandelbrot = nothing
193 img = CUDA.zeros(RGB{Float64}, n, m)
194 println("Startet Drawing")
195 bench_zeich!(horizontal, vertical, maxValues, img, maximum(maxValues))
196 println(maximum(maxValues))
197
198 maxValues = nothing
199 img_cpu = zeros(RGB{Float64}, n, m)

```

```
189     img_cpu .= img
190     save(string(@__DIR__) * "/Pictures/gpu/analyse/
191 AnalyBuddhabrotmengewithZoomGPU$(zoom)ToPoint$(zoomPoint)WithIteration$(iteration)withResolution$(m)x$(n).png", img_cpu)
191 end
```