Berechnung der Mandelbrotmenge: Optimierung und Farbgebung

Manuel Schmitt
Fachb. Angewandte Informatik
Hochschule Fulda
Leipzigerstr. 123, 36037 Fulda, Deutschland
manuel.schmitt@cs.hs-fulda.de

Thomas Mott
Fachb. Angewandte Informatik
Hochschule Fulda
Leipzigerstr. 123, 36037 Fulda, Deutschland
thomas.mott@cs.hs-fulda.de

***Zusammenfassung*—Die Mandelbrotmenge ist ein mathematisches Phänomen, das auf Grund seiner einzigartigen Beschaffenheit, Darstellung und Komplexität auch außerhalb der Mathematik Aufsehen erregt. Das Ziel dieser Arbeit ist es, zu untersuchen, wie die Berechnung der Mandelbrotmenge optimiert und ihre Farbgebung beeinflusst werden kann. Diese Untersuchung basiert auf GPU-Berechnungen des Fraktals. Grundlage für die GPU-basierte Berechnung ist das Framework *TensorFlow*. Um die Beeinflussung der Farbgebung zu demonstrieren, kommen mathematische Parameter, wie etwa die Anzahl der Berechnungsiterationen oder Minimal- und Maximalwerte für die Farbskala, als Variablen zum Einsatz. Eine im Rahmen der Untersuchungen entwickelte Demo-Anwendung ist in der Lage, entsprechende Laufzeitberechnungen anzustellen sowie miteinander vergleichbare Bilder von Ausschnitten der Mandelbrotmenge zu erstellen. Die Gegenüberstellung dieser vergleichbaren Ergebnisse ist Kernpunkt dieser Ausarbeitung.**

I. Vorwort

Dieser Forschungsbericht handelt von der nach dem Mathematiker Benoît Mandelbrot benannten Mandelbrotmenge; einem Fraktal. Ein Fraktal (siehe Abbildung 1) ist ein Gebilde, dessen komplexe Struktur sich über alle Längendimensionen entfaltet und rekursiv konstruiert ist. Fraktale werden häufig als selbstähnlich bezeichnet, was bedeutet, dass sie exakt oder ähnlich wie Teile von sich selbst aussehen. Solche komplizierten Muster können mit bestimmten Farbgebungen als computergenerierte Bilder erstellt werden. Um die Berechnung der Mandelbrotmenge zu verstehen, ist ein grundlegendes Verständnis von komplexen Zahlen erforderlich. Die unten aufgeführte Formel (1) ist notwendig, um die für das Fraktal benötigten Zahlen zu errechnen. Sie wird wiederholt auf jedes Element eines Ausschnitts der komplexen Ebene angewendet. [3, S. 1]

$$Z_{n+1} = Z_n^2 + C \tag{1}$$

Letztendlich wird jedem Element des Ausschnitts der komplexen Ebene ein festgelegter Farbwert zugeordnet, wodurch ein einzigartiges Bild entsteht. Die aufwendige Berechnung des Fraktals kann durch Parallelisierung beschleunigt werden. Seine Darstellung kann durch mathematische Parameter beeinflusst werden.

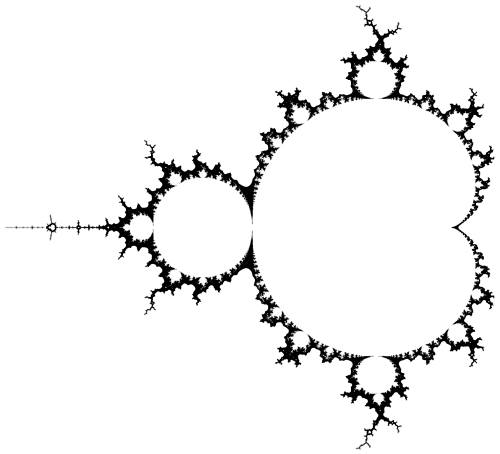

Abbildung 1. Silhouette der Mandelbrotmenge.

A. Projektübersicht

Dieses Projekt befasst sich mit der Untersuchung der Berechnung der Mandelbrotmenge in Bezug auf Laufzeitoptimierung und Beeinflussung ihrer Darstellung. Die eigens hierfür entwickelte Anwendung liefert entscheidende Ergebnisse, um Aussagen darüber zu treffen, unter welchen Umständen die Berechnung optimal verläuft sowie um Einflüsse bestimmter mathematischer Faktoren sichtbar zu machen. Laufzeitberechnungen unter verschiedenen Bedingungen können verglichen werden. Weiterhin können Bilder des Fraktals beziehungsweise von Ausschnitten der Mandelbrotmenge mit Hinblick auf optische Unterschiede untersucht werden.

B. Verwandte Arbeit

In dem im Jahr 2003 erschienenen Schriftstück *An overview of parallel visualisation methods for Mandelbrot and Julia set* von V. Drakopoulos, N. Mimikou und T. Theoharis [2] wird der Sachverhalt der parallelisierten Berechnung der bekannten Fraktale unter Berücksichtigung verschiedener algorithmischer Herangehensweisen beleuchtet [2, S. 2-3]. Parallelität der Algorithmen wird mit Hilfe des *Message Passing Interface (MPI)* hergestellt [2, S. 3]. Hierbei wird auch der Aspekt des Lastenausgleich zwischen einzelnen Prozessoren aufgegriffen [2, S. 5]. Das Resultat der Untersuchungen

ist, dass bestimmte Algorithmen schneller als andere sind, es aber auch zu Qualitätseinbußen bei den Ergebnisbildern kommen kann. Des Weiteren führt parallele Verarbeitung zu signifikanter Laufzeitverringerung, insbesondere bei optimaler Lastverteilung auf die Prozessoren [2, S. 11]. In dieser gegenwärtigen Ausarbeitung werden unterschiedliche Algorithmen zur Berechnung der Mandelbrotmenge oder der dabei mögliche Lastenausgleich zwischen Prozessoren nicht berücksichtigt. Diese Aspekte könnten aber in Zusammenhang mit einer *TensorFlow*-basierten Herangehensweise bei der Berechnung gebracht werden und somit als Grundlage für eine weiterführende Arbeit dienen.

Isaac K. Gangs, David Dobsons, Jean Gourds und Dia Alis Arbeit [3] aus 2007 betrachtet das Thema Parallelisierung der Berechnung der Mandelbrotmenge ebenfalls. Auch hier ist die Grundlage der Parallelität das *Message Passing Interface (MPI)* [3, S. 4]. Das Ergebnis ist, dass Parallelisierung durchaus effektiv ist und weiteres Bestreben zur Optimierung aussichtsreich sei [3, S. 7].

Im Unterabschnitt *Complex numbers and fractals* des zweiten Kapitels des Buches *Getting Started with TensorFlow* (2016) von Giancarlo Zaccone [9] wird die Berechnung des Fraktals unter Zuhilfenahme einer älteren Version des Frameworks *TensorFlow* veranschaulicht; zur Darstellung des Ergebnisbildes wird *Matplotlib* genutzt [9, S. 48-51]. Eine Implementation des häufig mit der Mandelbrotmenge in Zusammenhang gebrachten Julia-Menge wird hier ebenfalls gezeigt [9, S. 52-53].

Edmund Weitz erklärt in seiner Publikation *Konkrete Mathematik (nicht nur) für Informatiker* [8] (2018), wie die Mandelbrotmenge zustande kommt. Er veranschaulicht die Veränderung von Beispielelementen der komplexen Ebene durch iterative Anwendung der bekannten Formel (1) in einer Tabelle [8, S. 496]. Zudem wird aufgezeigt, wie sich die visuelle Form des Fraktals mit steigender Iterationsanzahl verändert und detaillierter wird [8, S. 497]. Ebenfalls erklärt wird, warum die Zahl 2 als Schranke für die Zugehörigkeit eines Elements der komplexen Ebene zur Mandelbrotmenge dient [8, S. 498-499]. Dies ist insofern besonders interessant, da diese Regel durch die für die gegenwärtige Ausarbeitung entwickelte Demo-Software aufgeweicht wird: Sie lässt die Auswahl eines beliebigen Schwellenwertes zu, um die dadurch entstehenden visuellen Veränderungen in den Ergebnisbildern zu zeigen.

C. Ziele

- Laufzeitoptimierung der Berechnung der Mandelbrotmenge. Das zugehörige Experiment ist in Abschnitt III-A zu finden.
  - Die nebenläufige Berechnung soll auf der GPU erfolgen.
  - *TensorFlow* ist die Bibliothek, die dies ermöglicht.
  - Laufzeiten sollen unter verschiedenen Bedingungen (*In-Place*, *Maskierung*) gemessen und gegenübergestellt werden (siehe Abbildung 5 und Tabelle I).
- Farbübergänge möglichst weich beziehungsweise wenig abrupt gestalten. Das Experiment in Abschnitt III-B nimmt sich dieser Thematik an.
  - Der Einfluss des Logarithmus auf Farbverläufe im Fraktal soll untersucht werden (siehe Tabelle II).
  - Dieser soll durch Berechnung von Varianzen in Ergebnisbildern messbar gemacht werden.

II. Übersicht der Methodik

Zur Umsetzung der Demo-Anwendung sind einige Vorkenntnisse zu unterschiedlichen Bibliotheken nützlich. Hierzu zählt beispielsweise Python mit den Modulen Matplotlib, NumPy und TensorFlow. Letzteres benötigt Zugriff auf GPUs um seine Performanz zu verbessern. Somit ist auch ein kurzer Einblick in CUDA sinnvoll. Abschließend wird ein Überblick in den eigentliche Aufbau der Demo-Anwendung und dessen Fallstricke aufgezeigt.

A. Python

Python ist eine interpretierte, interaktive, objektorientierte Programmiersprache die Module, *Exceptions*, dynamische Typisierung und Datentypen sowie Klassen umfasst. Zudem werden verschiedene Programmierparadigmen wie sowohl Objektorientierung als auch prozedurale und funktionale Programmierung unterstützt. Sie besitzt Schnittstellen zu vielen Systemaufrufen, Bibliotheken, Fenstersystemen und ist in C oder C++ erweiterbar. Durch die gute Portabilität läuft sie auf viele Unix-Varianten, einschließlich Linux und macOS, und auf Windows. Die Sprache ist außerdem Quelloffen (open-source) und kostenfrei nutzbar. [7, S.11]

B. Matplotlib

Matplotlib ist eine Bibliothek zur Erstellung von 2D *Plots* durch *Arrays* in Python. Sie zielt darauf ab, Grafiken in Publikationsqualität zu erzeugen. Die Erstellung von Diagrammen soll möglichst einfach sein und auch eine Erweiterbarkeit ist sichergestellt. Um dies zu gewährleisten, emuliert Matplotlib die Befehle von MATLAB und dessen *Plotting*-Fähigkeiten. Es werden zudem viele verschiedene Dateiformate wie SVG oder PNG für den Bildexport bereitgestellt. [5]

C. NumPy

NumPy ist die meistgenutzte Array-Programmier-Bibliothek für Python. Sie spielt eine essentielle Rolle in vielen Forschungsfeldern und stellt eine mächtige, kompakte und aussagekräftige Syntax zum Zugriff, zur Manipulation und für Operationen auf Vektoren, Matrizen und Arrays mit vielen Dimensionen bereit. Viele andere Bibliotheken, wie das zuvor genannte Matplotlib und auch Tensorflow, setzen intern NumPy ein. NumPy-Arrays sind das de facto Austauschformat für Array-Daten in Python. [4]

D. TensorFlow

TensorFlow stellt ein Python-Interface zur Verfügung, um Algorithmen für Machine Learning zu definieren und diese zu berechnen. Diese basieren zu großen Teilen auf linearer Algebra, also Vektoren und Matrizen. Somit können auch Berechnungen außerhalb von Machine Learning durchgeführt werden. Die Bibliothek unterstützt hierbei eine große Anzahl heterogener Systeme und erlaubt es, Arbeitsaufträge auf mehrere Maschinen zu verteilen. Kernelement bei der Berechnung ist die GPU-Untersützung (siehe CUDA). Diese beschleunigt durch hohe Parallelität die Vektor-, Matrix- und Array-Operationen. [1]

E. CUDA

Da GPUs hochgradig parallele *Multithreading*-Prozessorn mit sehr vielen Kerne (*Manycore*) sind, benötigen sie wegen ihrerer Architektur ein einheitliches Programmiermodell. Eines dieser ist die von NVIDIA entwickelte *Compute Unified Device Architecture* (CUDA). Sie ermöglicht die Nutzung von GPUs abseits der Grafikberechnung und erlaubt die einfache Entwicklung von Anwendungen mit hohem Anteil an Parallelverarbeitung. Grafikprozessoren können so Arbeitslasten durch Tausende von unabhängigen *Threads* gleichzeitig bearbeiten. Dabei skaliert das Modell transparent mit der Anzahl an Rechenkernen. CUDA ist nur mit von NVIDIA entwickelten GPUs lauffähig. [6, S. 42]

F. Implementation

„Die Mandelbrot-Menge $\mathbb{M}$ ist [...] definiert als die Menge aller komplexen Zahlen c, für die die Folge $(z_{c,n})_{n\in\mathbb{N}}$ beschränkt ist.“ [8, S. 496] Auf visueller Ebene betrachtet bedeutet dies, dass zwei Farben (bspw. Schwarz und Weiß) für die Darstellung des Fraktals ausreichend sind, da es prinzipiell nur zwei Zustände für einen Bildpunkt gibt: zur Menge zugehörig oder nicht. Die für dieses Projekt verwendete Implementierung der Menge nutzt für ihre Farbgebung eine eigene Palette (siehe Abbildung 2), die mit Schwarz beginnt und mit Rot endet.

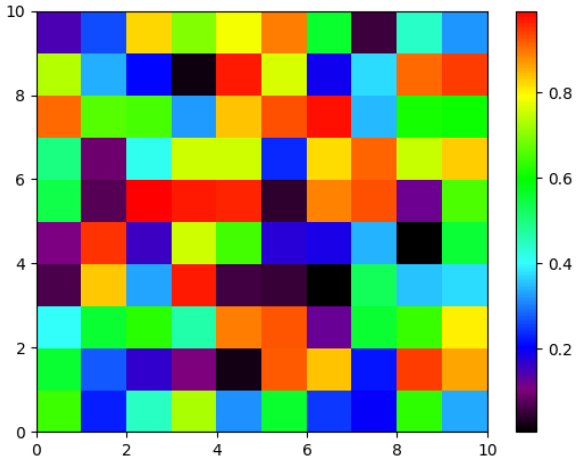

Abbildung 2. Farbpalette der Demo-Anwendung.

Dazwischen befinden sich Lila, Blau, Grün und Gelb. Die Demo ermöglicht es zudem, den Wertebereich der Farbpalette beliebig anzupassen und neu zu normalisieren und somit einen Teilbereich der ursprünglichen Skala zu einer neuen Palette zu machen, die dann dem aktuellen Ausschnitt des Fraktals seine Farbe gibt. Die dadurch entstehende Verschiebung des Farbraums kann einem Bild eine völlig neue Erscheinung geben. Bei der initialen Berechnung des Fraktals wird ein maximaler Wert für den Farbraum festgelegt, der sich nie ändert; auch nicht, wenn ein Teilausschnitt zu sehen ist, der diesen maximalen Wert nicht beinhaltet. Dies kann durch Einschalten der Option *Automatic Maximum* insofern verändert werden, dass für jeden Ausschnitt, ein neues Maximum gesucht wird. Diese Option kann das Erscheinungsbild ebenfalls erneuern, allerdings nur, wenn das Maximum des neuen Ausschnitts sich vom Maximum des letzten unterscheidet; was eher am Rand des initialen Ergebnisbildes der Fall ist als in der Mitte. Anders als bei der binären farblichen Gestaltung in Schwarz und Weiß (wie gezeigt von Weitz [8, S. 497-498]) werden somit mehrere verschiedene Farben verwendet, um dem Fraktal seine Erscheinung zu verleihen. Es wird demnach nicht nur unterschieden, ob ein Bildpunkt zur Mandelbrotmenge gehört oder nicht, sondern es wird pro Element der komplexen Ebene die Anzahl der Iterationen gespeichert, die nötig ist, bis ein Element divergiert. Falls ein Element nicht divergiert, wird somit die Anzahl der Iterationen als Wert angenommen. Dies ist notwendig, um ein breiter gefächertes Spektrum an Werten zu erhalten, sodass die Farben der Palette darauf abbildbar sind. Die daraus resultierende Matrix aus natürlichen Zahlen ist die Grundlage der finalen Farbgebung des jeweiligen Bildausschnitts. Wird die Option *Use Logarithm* genutzt, wird auf jeden Wert dieser Matrix der Logarithmus Naturalis angewendet und eine Matrix aus reellen Zahlen entsteht. Durch diesen zusätzlichen Rechenschritt, verringert sich die Farbvarianz innerhalb eines Bildausschnitts. Die geringere Varianz pro Ausschnitt bewirkt, dass Farbverläufe weicher erscheinen. Damit ist gemeint, das Übergänge von einer Farbe in die nächste nicht so abrupt auftreten und Nuancen besser zu erkennen sind. Dieser Effekt wird in Abschnitt III-B näher erörtert.

Die gesamte Mandelbrotmenge befindet sich in einem Bereich der komplexen Ebene, der auf der *reellen Achse* von -2 bis +2 und auf der *imaginären Achse* von -2i bis +2i verläuft [8, S. 496]. Die *reelle Achse* kann als *x-Achse* und die *imaginäre Achse* als *y-Achse* verstanden werden. Um verschiedene Ausschnitte des Fraktals betrachten zu können, ist es möglich, einen Teilausschnitt des jeweils momentan angezeigten Ausschnitts per *Rectangle Selector* auszuwählen und diesen als neues Ausgangsbild festzulegen. Dieser Vorgang lässt sich theoretisch unendlich oft wiederholen, ist aber auf technischer Ebene durch die Rechenleistung des Computers begrenzt, denn die Dimensionen des schrumpfenden Ausschnitts der komplexen Ebene sind irgendwann derart klein, dass diese Zahlen nicht mehr errechenbar sind, da der Datentyp *complex128* bzw. *float64* in seiner Größe beschränkt ist.

Die entwickelte Demo-Anwendung kommt unterstützend zum Einsatz, um verschiedene Experimente und Untersu-

chungen vorzunehmen. Wichtiges Hilfsmittel hierbei ist die Bibliothek *TensorFlow*. Für die Darstellung der berechneten Mandelbrotmenge und zur Bereitstellung einer einfachen Benutzeroberfläche ist Matplotlib dienlich. Es stehen zwei implementierte Varianten für die Berechnung der Menge zur Verfügung. Die erste nutzt den von *TensorFlow* bereitgestellten Datentyp *complex128* für komplexe Zahlen. Hier werden die Rechenregeln transparent für Entwickler bereitgestellt. Die zweite Variante nutzt zwei als Tupel zusammengefasste *float64*-Werte. Wie in (1) zu sehen ist, muss Z_n bei jedem Durchlauf quadriert werden. Die Multiplikation komplexer Zahlen stellt sich wie folgt dar:

$$z_1 \cdot z_2 = (a + bi) \cdot (c + di) \tag{2}$$

Da die beiden Faktoren bei einer Quadrierung identisch sind, ändert sich die Formel wie folgt:

$$z^2 = (a + bi) \cdot (a + bi) \tag{3}$$

Dies ergibt die erste binomische Formel. Sie lässt sich mit komplexen Zahlen wie folgt auflösen:

$$\begin{aligned} z^2 &= (a + bi)^2 \\ &= aa + abi + abi + bibi \\ &= a^2 + b^2 i^2 + 2abi, \ wobei \ i^2 = -1 \\ &= a^2 - b^2 + 2abi \end{aligned} \tag{4}$$

Die Formel lässt sich dann sehr einfach durch die von *TensorFlow* bereitgestellten *Reduction*-Funktionen, *Broadcasting* sowie mit *Stacking* in Python nachbauen. Die Addition hingegen benötigt keine zusätzlichen Umformungen und kann Problemlos übernommen werden.

Zusätzlich zu diesen beiden typbezogenen Varianten besteht die Möglichkeit, die Berechnungen *in-place* durchzuführen. Hierbei wird kein zusätzlicher Speicherplatz genutzt, um Zwischenergebnisse von Berechnungsschritten abzulegen und nur die im Tensor enthaltenen Werte werden überschrieben. Ohne dieses Verfahren legt Tensorflow bei jeder Iteration einen neuen Tensor an, was möglicherweise zu Performanz- und Speicherproblemen führen kann. Eine Maskierung ist ebenfalls implementiert. Bei dieser werden Teile der Mandelbrotmenge aus der aktuellen Berechnung entfernt, wenn sie einen vorgegebenen Schwellenwert überschreiten, also divergieren. Das hat zur Folge, dass je nach Bildausschnitt die Bearbeitungsdauer potenziell beschleunigt wird. Zudem werden Überschreitungen des Wertebereichs von Datentypen vermieden, da diese bei der Potenzierung von divergierenden Punkten sehr schnell auftreten können.

Sowohl die unterschiedlichen Berechnungsarten, also Berechnung mit komplexen Zahlen oder mit einfachen Float-Werten, sowie die Option *In-Place* als auch Maskierung können beliebig miteinander kombiniert werden. Problematisch hierbei ist ein aktueller Fehler[1] in *TensorFlow*, der es nicht erlaubt, die Funktion *gather_nd* mit einem variablen Tensor (also tf.Variable) der komplexe Zahlen enthält, zu nutzen. Sie wird bei der Berechnung der Masken benötigt. Dies ist auch der Grund, warum zwei unterschiedliche Implementationen im Hinblick auf Typisierung existieren. Der Fehler ist durch einen Workaround gelöst, der den variablen Tensor zu einer Konstante konvertiert. Kommt zusätzlich das *In-Place*-Verfahren zur Anwendung, ist es sogar nötig, eine Kopie anzulegen. Der Ablauf ist in Abbildung 3 zu sehen.

Um in den Experimenten Laufzeiten analysieren zu können, kommt ein Python-Decorator zum Einsatz, der die Dauer einer Funktion mittels eines Performance-Counters misst. Dieser findet vorrangig zur Ermittlung der Zeitspanne bei der Berechnung der Mandelbrotmenge Anwendung, sodass die Auswirkung der unterschiedlichen Optionen und Parameter gegenübergestellt werden können.

Die Nutzeroberfläche ist mit *Matplotlib* realisiert. Neben der Ausgabe des aktuellen Bildausschnittes der Mandelbrotmenge stehen einstellbare Parameter zur Verfügung:

- **Komplexe Zahlen**: legt fest, ob die Berechnung mit Werten des Datentyps *complex128* oder *float64* (2 Werte) geschehen soll.
- **In-place-Berechnung**: legt fest, dass kein zusätzlicher Speicherplatz verwendet wird.
- **Maskierung**: legt fest, dass divergierende Punkte bei der Berechnung nicht mehr beachtet werden sollen.
- **Auflösung**: erlaubt das Spezifizieren der Höhe und Breite des auszugebenden Ausschnittes. Eine ungerade *Aspect-Ratio* wird nicht korrigiert.
- **Schwellenwert**: legt den Wert fest, ab dem ein Element der komplexen Ebene bei der iterativen Anwendung der Formel (1) als divergierend gilt.
- **Iterationsanzahl**: legt die Anzahl der Durchläufe fest, die bei der Berechnung von Z_n stattfinden.
- **Logarithmus**: legt fest, ob nach der Berechnung des Bildausschnitts der Logarithmus angewendet wird.
- **Min-Max-Werte**: Slider zum Festlegen des minimalen und maximalen Wertes, der bei der Normalisierung der Farbskala Anwendung findet.
- **Automatisches Maximum**: legt fest, ob das Maximum für die Normalisierung automatisch ermittelt werden soll.

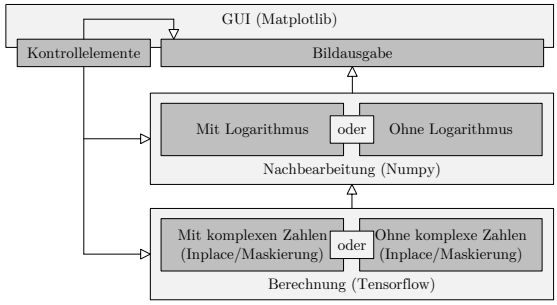

Abbildung 3. Grobablauf der Demo-Anwendung.

[1]Hierzu existiert der folgende Github-Issue: https://github.com/tensorflow/tensorflow/issues/59754

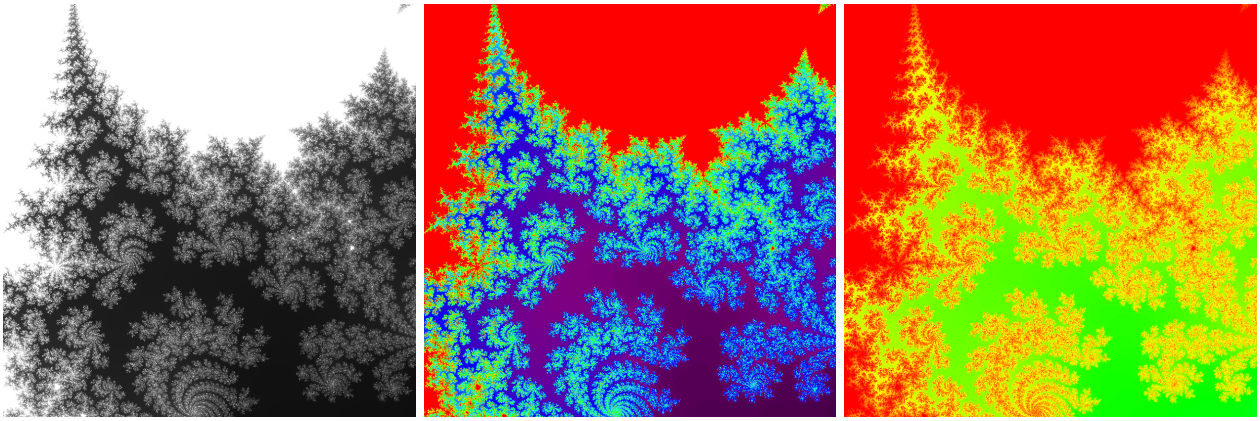

Abbildung 4. Gegenüberstellung eines Ausschnitts des Fraktals unter verschiedenen Bedingungen: Graustufenbild (links), farbiges Bild (mittig), farbiges Bild, das mit Logarithmus berechnet ist und einen weicheren Farbverlauf aufweist (rechts).

III. Experimente

In den folgenden Abschnitten sind die Experimente zu finden. Hier wird durch Zeitmessungen eine Analyse der einzelnen Laufzeiten und mittels Varianzen und Bildvergleichen eine Auswertung der Farbgebung durchgeführt. In Abbildung 4 ist ein erster Ausblick auf solche zu sehen.

A. Experiment 1: Analyse der Laufzeit

Im ersten Experiment wird die Laufzeit der beiden Implementationsvarianten (auch in Kombination mit den zusätzlich zur Verfügung stehenden Optionen) betrachtet. Zur Berechnung wird die folgende Hard- und Software verwendet:

- *CPU*: Intel i7-3930K
- *GPU*: NVIDIA GeForce RTX 3080
- *RAM*: 32GB
- *Betriebsystem*: Windows 11 mit WSL2 (Ubuntu)
- *Software*:
  - Python 3.9.13
  - TensorFlow 2.11.0
  - NumPy 1.23.4
  - Matplotlib 3.6.2
  - CUDA 12.1
  - GPU-Treiber: 531.29

Als Parameter, die bei jeder Messung genutzt werden, dienen die nachfolgenden Werte:

- *Auflösung*: 800x800 Pixel
- *Schwellenwert*: 2.0
- *Ausschnitt*: [-2,2] auf beiden Achsen.

Jede Laufzeitberechnung besteht aus drei Messungen und dem resultierenden Mittelwert. Hierbei wird mit einer Iterationsanzahl von 1000 begonnen und Schrittweise um weitere 1000 bis zum Maximalwert von 10000 erhöht. Es ergeben sich zehn Messpunkte bestehend aus Mittelwerten für jede Variante und jede Option. Somit erhält man insgesamt 80 nutzbare Datenpunkte. Diese berechneten Werte sind in Abbildung 5 zu sehen. Direkt auffällig ist, dass sich die Laufzeit mit der Iterationsanzahl nahezu linear verhält. Leichte Abweichungen lassen sich dadurch erklären, dass neben *TensorFlow* und *CUDA* auch anderen im Hintergrund des Betriebssystems laufende Anwendungen potenziell auf die GPU zugreifen und diese nutzen. Auch die eigentliche Jobverteilung auf der Hardware selbst kann zu leichten Variationen bei den Messungen führen. Die zweite Auffälligkeit ist, dass die eigene Implementation (*2xfloat64*) die interne *complex*-Implementation von TensorFlow in allen Punkten schlägt. Hier liegt die Vermutung nahe, dass die optimierte Variante der Multiplikation der komplexen Zahlen wie sie in Abschnitt II-F erläutert ist, zu diesem Performanzvorteil führt. Die Kombination *complex128* mit aktivierter *In-Place*- und Maskierungs-Option ist im Gegensatz zu allen anderen Varianten die langsamste und wird zum Zwecke der Übersichtlichkeit im Diagramm abgeschnitten. Aufgrund des eingesetzten Workarounds kann nicht bestimmt werden, ob das hier dargestellte Ergebnis verfälscht ist. Vergleicht man nun die Nutzung von keiner Option mit der Verwendung von *In-Place*, so ist zu erkennen, dass sowohl bei *complex128* als auch *2xfloat64* keine Performanzvor- oder -nachteile entstehen. Die Maskierung liefert bei beiden Varianten gegensätzliche Ergebnisse. Die *complex128* Implementation profitiert sehr stark von dieser, wohingegen die *2xfloat64* langsamer wird. Schaltet man zusätzlich noch die *In-Place*-Option hinzu, so verändert sich, wie erwartet, ebenfalls nichts.

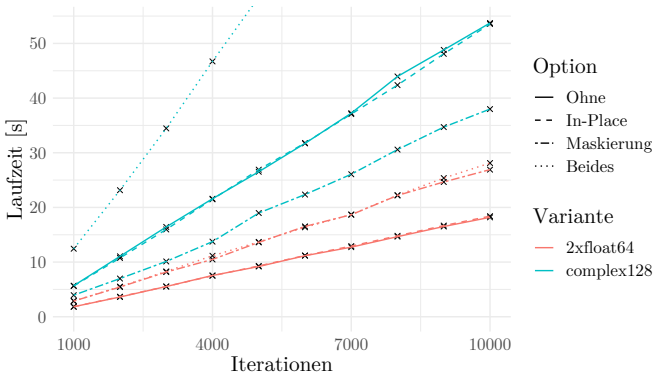

Abbildung 5. Zeitmessungen nach Varianten und Optionen.

Tabelle I
DURCHSCHNITTLICHE BERECHNUNGSDAUER NACH VARIANTE UND OPTIONEN IN SEKUNDEN

Variante \ Option	Ohne	In-Place	Maskierung	Beides
complex128	53.7370	53.5540	37.9887	119.4969[a]
2xfloat64	18.1862	18.4007	26.9152	28.1174

[a]Hinweis: Hier wird ein Workaround genutzt.

In Tabelle I sind die Laufzeiten mit einer Iterationsanzahl von 10000 als numerische Werte zu sehen. Sie verdeutlichen die Aussagen zu Abbildung 5.

B. Experiment 2: Analyse der Farbgebung

Für dieses Experiment wird eine Vielzahl von Bildern von Ausschnitten des Fraktals stichprobenartig herangezogen. Es wird stets ein und der selbe Bildausschnitt ohne und unter Verwendung des Logarithmus verglichen. In dieser Ausarbeitung werden fünf Paare von Vergleichsbildern dargestellt (siehe Abbildungen 6, 7, 8, 9, 10, 11, 12). Das Ergebnis ist eindeutig: Die Farbgebung des Fraktals kann durch Verwendung des Logarithmus (Naturalis) derart beeinflusst werden, dass Farbverläufe deutlich weicher und viel weniger abrupt sind. In einigen Fällen macht sich das sogar dadurch bemerkbar, dass das sichtbare Farbspektrum des Vergleichsbildes, das unter Verwendung des Logarithmus erstellt wurde, kleiner ist als das des Bildes, das ohne seine Verwendung erstellt wurde (siehe Abbildungen 8, 9, 10, 11, 12). Die Auswirkungen des Logarithmus können allerdings nicht nur durch scharfes Hinsehen erkannt werden. Sie können auch durch Messung der Varianz des jeweiligen Bildausschnitts bestätigt werden. Die Ergebnisbilder dieses Experiments, welche mit Logarithmus berechnet sind, weisen stets eine geringere Varianz auf als solche, die ohne berechnet sind. Dies wird durch Tabelle II verdeutlicht.

Tabelle II
VARIANZEN DER VERGLEICHSBILDER (WERTEBEREICH [0, 1]). ALLE BILDER SIND MIT 1000 ITERATIONEN UND EINER AUFLÖSUNG VON 800X800 PIXELN ERSTELLT.

Bildreferenzen	Varianz ohne Log.	Varianz mit Log.	Veränderung
Abb. 6 + 7	0.0849	0.0761	-10.4%
Abb. 8	0.2073	0.0878	-57.6%
Abb. 9	0.1368	0.0531	-61.2%
Abb. 10	0.1736	0.0386	-77.8%
Abb. 11 + 12	0.2275	0.0718	-68.4%

IV. DISKUSSION

In Bezug auf das Experiment in Abschnitt III-A ist festzuhalten, dass der Algorithmus erfolgreich mittels *TensorFlow* auf der GPU ausgeführt wird. Die Ermittlung und Gegenüberstellung der Laufzeiten unter den in Tabelle I aufgeführten Optionsmöglichkeiten (*ohne*, *In-Place*, *Maskierung*, *Beides*) ist demnach voll bewerkstelligt. Es muss allerdings hinzugefügt werden, dass keine Rückschlüsse darauf gezogen werden können, wie die Laufzeit unter der Kombination *complex128*, *In-Place*, *Maskierung* ohne das genannte Problem in der aktuellen *TensorFlow*-Version ausfallen würde.

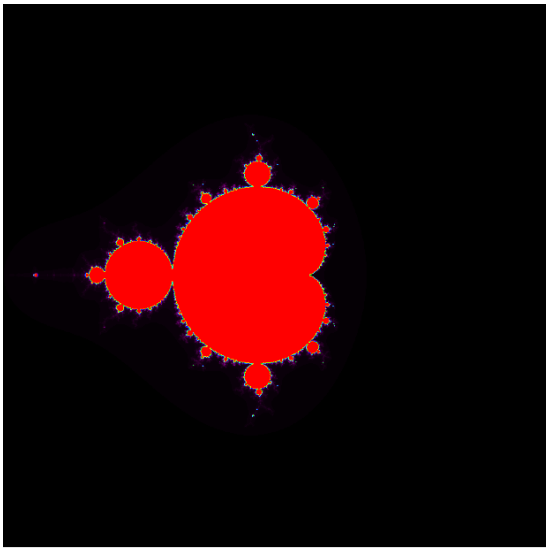

Abbildung 6. Initialer Ausschnitt ohne Logarithmus.

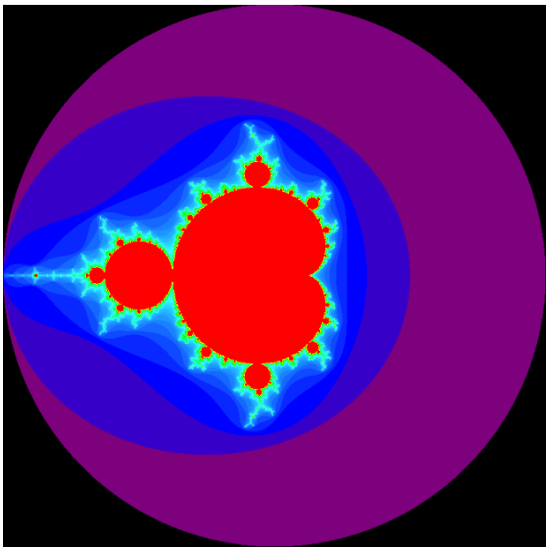

Abbildung 7. Initialer Ausschnitt mit Logarithmus.

Das Experiment in Abschnitt III-B ist erfolgreich verlaufen. Das Ziel, weichere Farbverläufe durch Miteinbeziehen des Logarithmus in die Berechnung zu erhalten, ist vollumfänglich erreicht worden, was in Tabelle II klar ersichtlich wird. Die im Rahmen dieses Experiments gemessenen Varianzen unter Nutzung des Logarithmus sind allesamt kleiner als jene, die ohne seine Nutzung gemessen sind.

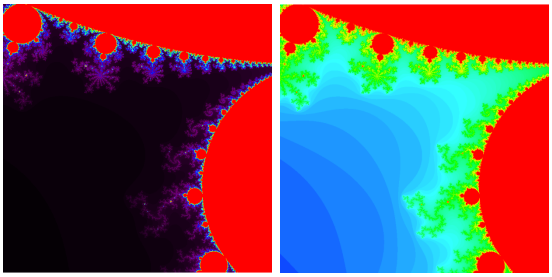

Abbildung 8. Testausschnitt 1: links ohne und rechts mit Logarithmus.

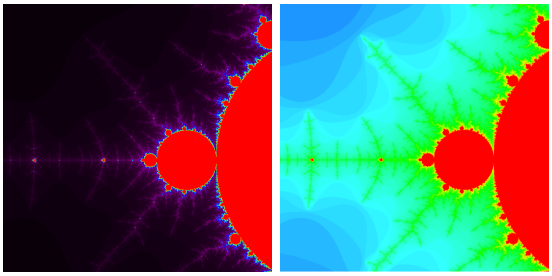

Abbildung 9. Testausschnitt 2: links ohne und rechts mit Logarithmus.

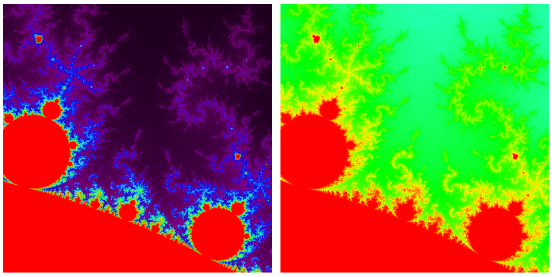

Abbildung 10. Testausschnitt 3: links ohne und rechts mit Logarithmus.

V. Schlussfolgerungen und Ausblick

- Laufzeitoptimierung der Berechnung der Mandelbrotmenge:
  - Die Laufzeit der *In-place*-Berechnung der Menge unter Verwendung des Frameworks *TensorFlow* unterscheidet sich nahezu nicht von der Laufzeit der *Out-of-place*-Berechnung.
  - Die Verwendung von Masken bewirkt bei der Implementierung mit dem Datentyp *complex128* eine signifikante Laufzeitverbesserung von rund 30%. Bei der Implementierung mit dem Datentyp *(2x)float64* ist eine Laufzeitverschlechterung von bis zu circa 30% zu verzeichnen.
  - Werden die *In-Place-* und *Maskierungs*-Option bei der Berechnung kombiniert, so tritt mit dem Datentyp *complex128* eine signifikante Verschlechterung von knapp 120% auf, was sehr wahrscheinlich auf den erwähnten Fehler im Framework zurückzuführen ist. Bei der Berechnung mit dem Datentyp *(2x)float64* ist -wie bei der Maskierung- eine Verschlechterung um rund 30% eingetreten.
- Beeinflussung der Farbgebung des Fraktals:
  - Wird der Logarithmus bei der Berechnung der Mandelbrotmenge genutzt, so werden Farbverläufe deutlich weicher dargestellt (siehe Abbildungen 6, 7).
  - Dies ist durch die Gegenüberstellung der Varianzen der Ergebnisbildausschnitte untermauert. Bei allen im Rahmen dieses Projekts verglichenen Testbildern ist eine eindeutige Tendenz erkennbar: Die Nutzung des Logarithmus verringert die Varianz.

Falls der Fehler in der aktuellen *TensorFlow*-Version irgendwann behoben sein sollte, wären die Laufzeitberechnungen mit der Kombination *complex128*, *In-Place*, *Maskierung* unter den dann korrekten programmatischen Bedingungen zu wiederho-

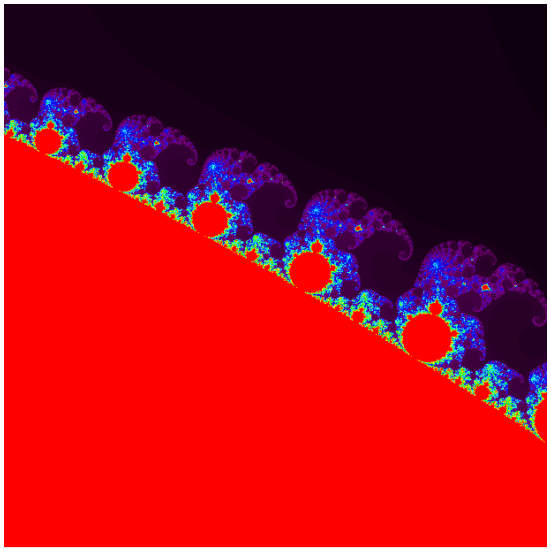

Abbildung 11. Testausschnitt 4 ohne Logarithmus.

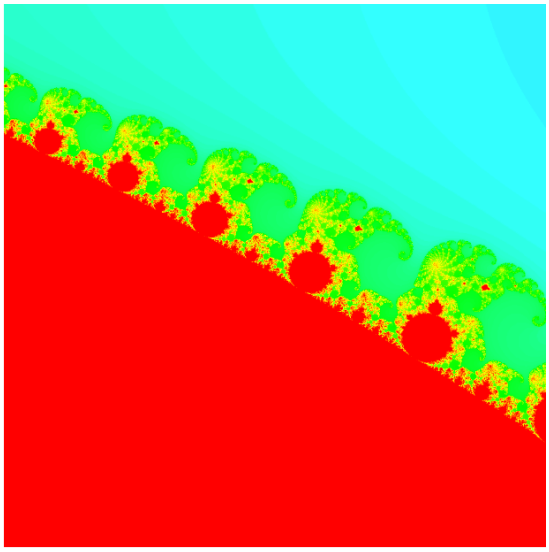

Abbildung 12. Testausschnitt 4 mit Logarithmus.

len, um festzustellen, ob sich an der bisherigen Laufzeit etwas ändern würde.

Ebenfalls interessant für eine weiterführende Arbeit wäre es, die in der Arbeit von V. Drakopoulos, N. Mimikou und T. Theoharis [2] genutzten, für bessere Laufzeit optimierten Algorithmen zur Berechnung der Mandelbrotmenge unter Verwendung von *TensorFlow* neu zu implementieren und somit GPU- statt CPU-basierte Laufzeitberechnungen anstellen zu können.

Weiterhin ist vorstellbar, dass auch andere mathematische Operationen die Erscheinung und Farbgebung des Fraktals beeinflussen können.

Außerdem sind alle genannten Untersuchungen nicht nur mit der Mandelbrotmenge, sondern auch mit den Julia-Mengen durchführbar.

LITERATUR

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[2] V. Drakopoulos, N. Mimikou, and T. Theoharis. An overview of parallel visualisation methods for Mandelbrot and Julia sets. Technical report, University of Athens, Panepistimioupolis 157 84, Athens, Greece, 2003.

[3] I. Gang, D. Dobson, J. Gourd, and D. Ali. Parallel Implementation and Analysis of Mandelbrot Set Construction. Technical report, University of Southern Mississippi, Hattiesburg, MS 39406-5106, 2008.

[4] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, Sept. 2020.

[5] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[6] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, mar 2008.

[7] M. Weigend. *Python GE-PACKT*. mitp Verlags GmbH & Co. KG, 2015.

[8] E. Weitz. *Konkrete Mathematik (nicht nur) für Informatiker*. Springer Spektrum, 2018.

[9] G. Zaccone. *Getting Started with TensorFlow*. PACKT PUBLISHING, 2016.

ANHANG A: INSTRUKTIONEN

- Repository:
  https://gitlab2.informatik.hs-fulda.de/fdai4965/mandelbrotresearch.git
- Genutztes Betriebssystem:
  Windows 10+ (mit WSL2), Linux
- Genutzte Software:
  - Python 3.9
  - TensorFlow 2.11
  - NumPy 1.23.4 (wird mit TensorFlow installiert)
  - Matplotlib 3.6.2
  - (Optional) Funktionierende CUDA-Installation
- Benutzung: Repository klonen und mit Python die Datei *app.py* ausführen.