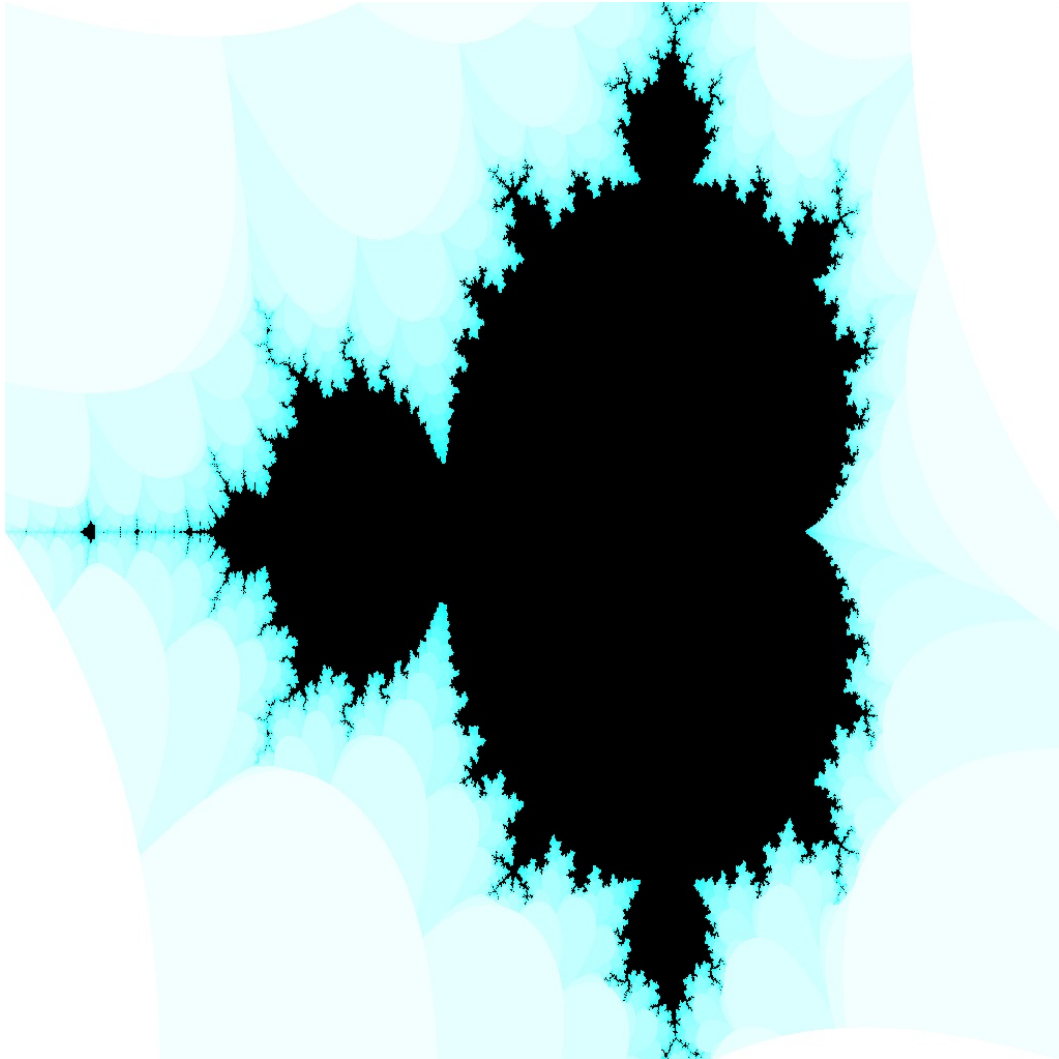


LEHRSTUHL FÜR RECHNERTECHNIK UND RECHNERORGANISATION
**Aspekte der systemnahen Programmierung
bei der Spieleentwicklung**

Projektaufgabe - 1: Mandelbrotmenge

Robin Ostner 
Oliver Jung
Florian Sprang



Inhaltsverzeichnis

1	Problemstellung und Spezifikation	3
2	Lösungsalternativen	4
3	Dokumentation der Implementierung	4
3.1	Implementierung Single-Pixel-Calculation	5
3.2	Implementierung Mult-Pixel-Calculation	6
3.2.1	Konzept	6
3.2.2	Konkrete Umsetzung	7
3.3	Anwendung des Programmes	9
4	Ergebnisse / Fazit	12
4.1	Ergebnisse und Optimierung	12
4.2	Fazit	13

1 Problemstellung und Spezifikation

2 Lösungsalternativen

3 Dokumentation der Implementierung

3.1 Implementierung Single-Pixel-Calculation

3.2 Implementierung Mult-Pixel-Calculation

Ein grundlegender Teil unserer Aufgabenstellung, war es das Programm mittels der SIMD-Einheit NEON, die auf dem Raspberry-Pi verbaut ist, zu optimieren. Hierbei arbeiteten wir zunächst an einem Ansatz, welcher leicht umzusetzen war jedoch nicht die erwünschte Effizienz bot. Unser erster Ansatz basierte darauf zuerst von allen Punkten, welche wir später zu berechnen hatten, den Real- bzw. Imaginärteil in ein float-Array zu speichern. Anschließend wollten wir dieses verwenden um mittels der strukturierten Ladefunktion von NEON vier Pixel gleichzeitig zu laden um diese anschließend zu berechnen. Allerdings war unser Verständnis für das Programmieren von ARM-Prozessoren zu diesem Zeitpunkt noch nicht ausreichend um unsere Idee vollständig umzusetzen. Während der Lösungsfindung für den ersten Versuch fingen wir an uns mit wichtigen Konzepten vertraut zu machen welche sich nicht direkt auf das reine Code schreiben beziehen. Um unsere Fehler schneller zu finden eigneten wir uns Wissen über den gcc-Debugger an, welches sich später noch als großer Vorteil erweisen sollte. Für unseren zweiten Lösungsansatz griffen wir erneut ein Prinzip auf, welches bereits zuvor einmal von einem unserer Teammitglieder angesprochen wurde, jedoch schnell wieder vergessen wurde da wir uns einig waren, dass dieses zu schwer umzusetzen sei. Im weiteren Verlauf wird auf das Konzept hinter diesem Prinzip und unsere tatsächliche Implementierung genauer eingegangen. Außerdem werden einige unserer Probleme geschildert und welche Ansätze wir verfolgt haben um diese zu lösen.

3.2.1 Konzept

Unser zweiter Versuch basiert auf dem Prinzip eine dauerhafte Berechnung aufrecht zu erhalten. Wie bereits zuvor erwähnt fällt ein Pixel unter 2 Bedingungen aus der Berechnung heraus. Erstens wenn er eine maximale Anzahl an Schleifendurchläufen mitgemacht hat, oder zweitens wenn gilt:

$$(\text{Imaginärteil} + \text{Realteil})^2 > \text{Schwellwert}$$

Diese dauerhafte Berechnung wird nur dann Unterbrochen wenn ein Pixel eines seiner Abbruchkriterien erfüllt. Tritt dieser Fall ein besitzt jeder Pixel eine eigene Funktion, welche den aktuell berechneten Pixel in das Bild für das Array sichert und anschließend aus den Konstanten festgelegten Registern die neuen zu berechnenden Werte holt, sowie die Speicherposition für den nächsten Pixel. Außerdem wird auch noch sein Laufwert zurückgesetzt, da wir diesen benutzen um den Grad der Konvergenz zu ermitteln und die Korrespondierenden Pixel blau zu färben. Des Weiteren werden alle Werte für den neuen Pixel berechnet und gespeichert. Hierbei war die Schwierigkeit einen Zeilenumbruch in den Pixelreihen zu erkennen und korrespondierend dazu die Laufwerte zu aktualisieren. Mittels Abbildung 1 soll das gerade beschriebene Konzept noch einmal vereinfacht dargestellt werden. Im weiteren Verlauf wird nun auf die konkrete Implementierung unseres Konzeptes eingegangen.

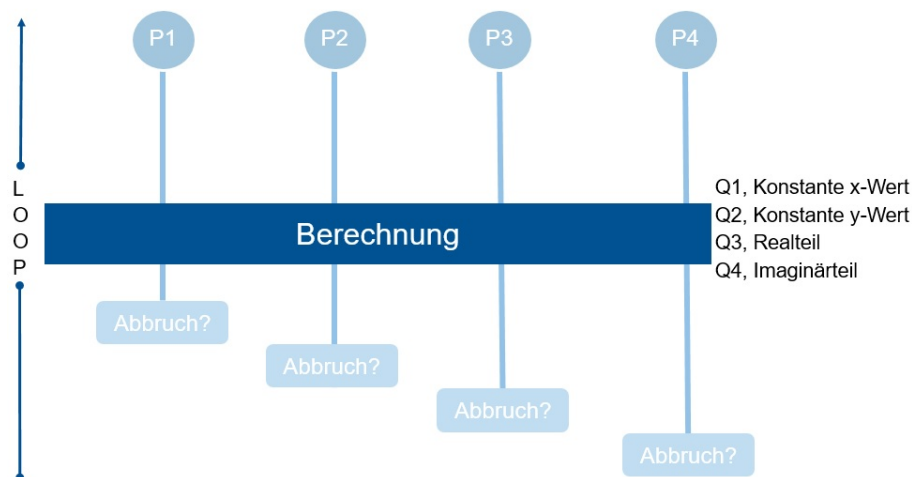


Abbildung 1: Grafische Darstellung des Berechnungsprinzips

3.2.2 Konkrete Umsetzung

Bei unserer Umsetzung der Mandelbrotmenge benutzen wir alle Ressourcen die uns der Prozessor zu Verfügung stellt. Konkret bedeutet das, dass wir alle allgemeinen Register sowie das Link Register benötigen und deshalb alle Register auf dem Stack sichern. Da wir mehr als vier Übergabewerte haben, werden die letzten beiden Werte vom Stack geladen. Der Anfang zu der Single-Pixel-Calculatation hat sich jedoch nicht großartig verändert. Zu Beginn berechnen wir die Gesamtlängen, welche wir pro Achse durchlaufen müssen. Mittels der Auflösung wird anschließend der Unterschied zwischen zwei Pixeln in der FPU berechnet. Hierbei ist zu beachten, dass wir von der Auflösung zuvor eine Kopie erstellt haben um von dieser eins abzuziehen, da ansonsten immer der letzte Pixel einer Reihe nicht auf dem Endpunkt liegt sondern einen Abstand davor. Das Register q0 wurde anschließend als Register für Variablen festgelegt, welche während der konstanten Berechnung benötigt werden. Die freien allgemein Register werden nun für die Hauptberechnung initialisiert. Da zu Beginn festgelegt wurde, dass die Größe des Bildes immer $Auflösung^2$ ist, wird in Register zwei diese gespeichert. Register zwei ist also der Schleifenlaufwert, welcher verwendet wird um festzustellen wann alle Pixel berechnet sind. Die weiteren Register drei bis sechs enthalten die Anzahl der bereits durchlaufenen Iterationen. Die Maximale Anzahl an erlaubten Iterationen wird in dem darauffolgendem Register R7 gespeichert. Somit ist es später möglich mit diesen Registern das bereits oben erwähnte Abbruchkriterium der maximalen Iterationsdurchläufe zu realisieren. Die Register R8-R11 werden nun mit der für den Pixel entsprechenden Speicheradresse initialisiert. Hierbei ist zu beachten, dass R0 immer den Wert enthält für den nächsten Pixel nicht für den letzten. Es werden vier Separate Speicher deshalb verwendet, da während der Berechnung die Pixel nicht immer gleich lang brauchen und somit Pixel4 vor Pixel1 fertig sein kann. Würde man eine einzelne Speicheradresse

dafür verwenden würde die Reihenfolge der berechneten Pixel im Bildarray nicht mehr stimmen. Um das Problem zu lösen müsste Pixel4 bei schnellerer Berechnungszeit auf die Berechnung von Pixel1 warten, was viel Zeit kosten kann. Um die Vorbereitungen für die Hauptberechnung abzuschließen werden die beiden float-Register s30 und s31 initialisiert. In s30 steht der aktuelle x-Wert, also der Wert der zuletzt verwendet wurde und in s31 wurde der Schwellwert geschrieben. Die Tabelle 1 und Tabelle 2 zeigen eine Übersicht über die allgemein und floating-point Register während der Berechnung.

Damit die Hauptschleife zu Beginn keinen Leerdurchlauf durchführt, werden vor Beginn der Schleife die entsprechenden Register (siehe: Abbildung 1) mit den ersten vier Pixelkoordinaten befüllt. Damit es zu keinen Fehlern in der maximalen Anzahl an zu berechnenden Pixeln gibt, wurden bereits nach der Berechnung von R2 diese vier Pixel abgezogen. Demnach stehen jetzt in q1 und q2 die x-Werte bzw. y-Werte. Somit bilden q1, q2 unser Konstante. Da die erste Iteration der Schleife als Ergebnis die Konstante selber ist kann man diese überspringen, indem man in die Register q3 (Realteil von z_i) und q4 (Imaginärteil von z_i) direkt die Konstante kopiert. Das Überspringen der ersten Iteration wird auch bei einem Pixelwechsel umgesetzt. Dadurch lässt sich pro Pixel eine Iteration einsparen, was sich später positiv auf die Laufzeit auswirkt.

Zu diesem Zeitpunkt wird nun die Schleife gestartet. Um die binomische Formel von $(z_i)^2 = (a * i + b)^2 = b^2 - a^2 + 2 * a * b$ zu realisieren wird q5 als Zwischenspeicher von $2 * a * b$ verwendet. Wir berechnen $2 * a * b$ zuerst, da wir dadurch nur ein Register als Zwischenspeicher benötigen. Somit können die Register die Real- und Imaginärteil beinhalten anschließend quadriert werden. Gemäß der Formel werden nun die neuen Real- und Imaginärteile berechnet. Damit ist die eigentliche Berechnung von einer Iteration zu Ende.

Der weitere Teil an parallelen Schritten innerhalb von NEON dient dazu das zweite Abbruchkriterium zu testen. In q5 werden nun Real und Imaginärteil aufsummiert und anschließend das Quadrat der Summe in q6 berechnet. Hier wird das Quadrat verwendet, da somit nicht auf Divergenz von negative Zahlen getestet werden muss. Der Schwellwert wurde zuvor so ausgewählt, das dieser bereits das quadratische des eigentlichen Schwellwertes ist. Im weiteren Verlauf erfolgen nun die sequentiellen Überprüfungen der Pixel gemäß der Abbruchbedingungen. Da die Überprüfung der einzelnen Pixel vom Aufbau her gleich ist wird hier nur auf die Überprüfung eines Pixels eingegangen. Die sequentielle Überprüfung aller vier Pixel ist allerdings notwendig, da in den gespeicherten Variablen in r0, q0 und q7 immer nur die Koordinaten für den nächsten Pixel stehen. Zuerst erfolgt die Überprüfung auf den Schwellwert, da dieser Fall häufiger eintritt und sich somit erneut ein paar Rechenoperationen einsparen lassen. Ist diese Überprüfung erfolgreich, springen wir in die Funktion `earlybailout` des jeweiligen Pixels. In dieser wird aufgrund der in R8-R11 gespeicherten Adressen die ersten beiden Bytes des Pixels mit dem *maximalenFarbwert* = 255, für weiß initialisiert. Um nun den Grad der Divergenz zu erhalten wenden wir folgende Formel an:

$$\text{zuspeichernderFarbwert} = \text{maxFarbwert} - (12 * \text{Anzahl der Schleifendurchläufe})$$

Die 12 ergibt sich aus dem maximalen Farbwert geteilt durch die maximale Anzahl

an Schleifendurchläufen. Das Ergebnis wird anschließend an die Stelle des dritten Byte gespeichert. Sollte jedoch die andere Abbruchbedingung eintreten, auf welche im Anschluss geprüft wird, so werden an die gespeicherte Speicheradresse der minimale Farbwert null gespeichert, was in einen schwarzen Pixel resultiert. Dies geschieht in der `maxBailout` Funktion des jeweiligen Pixels.

Wenn ein Pixel nun fertig berechnet und seine Werte in das Bildarray gespeichert wurden, muss er mit dem nächsten zu berechnenden Pixel ausgetauscht werden. Das Austauschen erfolgt in der jeweiligen `Reset` Funktion. In dieser wird zunächst der Laufwert auf -1 gesetzt. Der Laufwert wird nicht auf null gesetzt, da wir in der Hauptschleife erst nach allen Überprüfungen die Laufwerte inkrementieren und somit bei der ersten Iteration der neue Pixel mit dem Laufwert 0 beginnt. Der nächste Schritt ist die Überprüfung auf einen Zeilenum sprung. Da uns die Länge einer Zeile bekannt ist, passiert dann ein Zeilenumbruch wenn gilt:

$$restlicheAnzahlPixeln \% Auflösung = 0$$

Da die vorhandene ARM-Architektur keine passende Modulo Operation bietet und auch keine Division von Ganzzahlen unterstützt wird die standardmäßig vorimplementierte Funktion des gcc-Compiler für zwei unsigned Integer Werte verwendet. Mittels der `Multiply with Subtract` Instruktion kann man jetzt den Rest einer Division berechnen. Ist das oben genannte Kriterium erfüllt so findet ein Zeilenumbruch statt. Bei diesem werden der aktuelle y-Wert um einen Schritt erhöht und der aktuelle x-Wert auf `Start.X - xSchritt` gesetzt. Das setzen des x-Wertes ergibt sich daraus, dass wir im weiteren Vorgehen des initialisieren des neuen Pixels, den x-Wert standardmäßig um einen Schritt erhöhen. So ist gewährleistet, dass der neue Pixel bei `xStart` beginnt. Am Schluss wird nun die neue Konstante gespeichert und aus den bereits vorher erwähnten Gründen die Konstante auch in Real- und Imaginärteil kopiert. Auch der neue zu Speichernde Adressbereich wird aktualisiert.

Sobald eines der beiden Abbruchkriterien eintritt wird auf das nächste Kriterium hin nicht mehr überprüft. Sonst kann der noch nicht berechnete neue Pixel direkt gespeichert werden. Nachdem alle Pixel überprüft wurden erhöhen wir deren Laufwerte um eins und überprüfen ob wir noch weitere Berechnungen durchführen müssen. Der gerade geschilderte Ablauf passiert solange wir alle Pixel berechnet haben.

3.3 Anwendung des Programmes

Um jetzt die bereits detailliert beschriebene Implementierung der Mandelbrotmenge zu verwenden, muss man das Programm mittels des Makefile compilieren. Der Programmaufruf erfolgt dann mittels der Konsole mit folgenden Übergabeparametern:

xStart: Integerwert der den Anfangswert auf der x-Achse repräsentiert

xEnd: Integerwert der den Endewert auf der x-Achse repräsentiert

yStart: Integerwert der den Anfangswert auf der y-Achse repräsentiert

yEnd: Integerwert der den Endewert auf der y-Achse repräsentiert

Auflösung: Integerwert für den gelten muss $Auflösung = 2^n$ für $n > 2$ 

Ein konkreter Aufruf wäre `./main -2 1 -1 1 1024`. Für die jeweiligen **Start bzw End werte** muss gelten, dass $Start < End$. Wird bei der Auflösung für n ein Wert größer als 12 gewählt, wird das Bild berechnet jedoch steigt die Zeit, die dafür benötigt wird, sehr schnell an. **Während** man das Programm laufen lässt, kann man sich nun entscheiden, ob man die Single-Pixel-Calculation verwendet oder die parallelisierte Version. Am Ende erhält man eine Datei des .bmp Formates. Diese kann man sich jetzt mit dem bevorzugten Programm für Bilder anschauen.

R0	Speicheradresse zum nächsten Pixel
R1	Auflösung
R2	Anzahl der Restlichen zu berechnenden Pixel
R3	Durchlaufene Iterationen Pixel 1
R4	Durchlaufene Iterationen Pixel 2
R5	Durchlaufene Iterationen Pixel 3
R6	Durchlaufene Iterationen Pixel 4
R7	Maximale Anzahl an erlaubten Iterationen
R8	Speicheradresse von Pixel 1
R9	Speicheradresse von Pixel 2
R10	Speicheradresse von Pixel 3
R11	Speicheradresse von Pixel 4
R12	allzweck Register zum berechnen oder zur Farbwertspeicherung

Tabelle 1: Einteilung der allgemein Register zur Berechnungszeit

q0	s0	Der Startwert auf der x-Achse
	s1	Abstand zwischen zwei Pixeln entlang der x-Achse
	s2	Aktuelle y-Position
	s3	Abstand zwischen zwei Pixeln entlang der y-Achse
q1	s4	Pixel 1 x-Position / Konstante Realteil
	s5	Pixel 2 x-Position / Konstante Realteil
	s6	Pixel 3 x-Position / Konstante Realteil
	s7	Pixel 4 x-Position / Konstante Realteil
q2	s8	Pixel 1 y-Position / Konstante Realteil
	s9	Pixel 2 y-Position / Konstante Realteil
	s10	Pixel 3 y-Position / Konstante Realteil
	s11	Pixel 4 y-Position / Konstante Realteil
q3	s12	Pixel 1 z_i Realteil
	s13	Pixel 2 z_i Realteil
	s14	Pixel 3 z_i Realteil
	s15	Pixel 4 z_i Realteil
q4	s16	Pixel 1 z_i Imaginarteil
	s17	Pixel 2 z_i Imaginarteil
	s18	Pixel 3 z_i Imaginarteil
	s19	Pixel 4 z_i Imaginarteil
q5	s20	Berechnung: Pixel 1: $2 * a * b$ / Realteil+Imaginärteil
	s21	Pixel 2: $2 * a * b$ / Realteil+Imaginärteil
	s22	Pixel 3: $2 * a * b$ / Realteil+Imaginärteil
	s23	Pixel 4: $2 * a * b$ / Realteil+Imaginärteil
q6	s24	Pixel 1: $(\text{Imaginärteil} + \text{Realteil})^2$ / Abbruchtest gegen Schwellwert
	s25	Pixel 2: $(\text{Imaginärteil} + \text{Realteil})^2$ / Abbruchtest gegen Schwellwert
	s26	Pixel 3: $(\text{Imaginärteil} + \text{Realteil})^2$ / Abbruchtest gegen Schwellwert
	s27	Pixel 4: $(\text{Imaginärteil} + \text{Realteil})^2$ / Abbruchtest gegen Schwellwert
q7	s28	unbenutzt
	s29	unbenutzt
	s30	Aktuelle x-Position
	s31	Schwellwert

Tabelle 2: Einteilung der NEON/FPU-Register zur Berechnungszeit

4 Ergebnisse / Fazit

4.1 Ergebnisse und Optimierung

Nachdem wir unsere parallele Implementierung fertiggestellt hatten, beginnen wir damit unsere beiden Programme zu vergleichen. Mit der implementierten Laufzeitanalyse wurden für alle zulässigen Auflösungen zwischen 8 und 8192 die Laufzeit von beiden Berechnungsarten zehn mal gemessen. Anschließend wurde der Mittelwert aus den zehn gestoppten Werten gebildet und miteinander verglichen. In Abbildung 2 wurde die Laufzeit der Single-Pixel-Calculation durch die der parallelen Berechnung geteilt um somit einen Faktor zu erhalten um wie viel schneller die Multi-Pixel-Calculation ist. Zunächst nahmen wir an, dass die einfache Berechnung in den niedrigen Auflösungsbereichen schneller als die parallele sei. Jedoch zeigte sich auch in diesen Bereich, dass die parallele Berechnung schneller ist. Jedoch würde man erwarten, dass sie circa drei mal so schnell sei, da wir anstelle von einem Pixel vier Pixel auf einmal bearbeiten. Diese Diskrepanz lässt sich dadurch erklären, dass die NEON-Einheit für die parallele Berechnung zusätzliche Schritte unternehmen muss um die Berechnungen parallel durchzuführen. Allerdings ist die durchschnittliche Verringerung der Laufzeit um den Faktor 1/2 in etwa das was am Ende zu erwarten war.

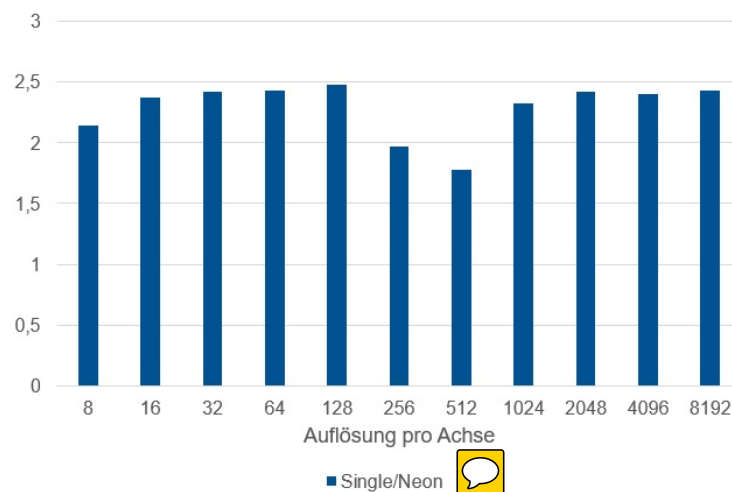








Abbildung 2: Vergleich von paralleler Berechnung zur einfachen Berechnung

Da wir bei unserer Implementierung eher den Ansatz verfolgt haben die Laufzeit zu minimieren, sind wir nicht speichereffizient, was die Register angeht. So finden sich in dieser Hinsicht einige Optimierungsmöglichkeiten. Als erster Ansatzpunkt wäre das Vereinigen der Pixelabbruchfunktionen. Hier liese sich sicherlich eine Möglichkeit finden auf Kosten von Laufzeit eine einheitliche Funktion zu schreiben. Außerdem könnte man auch Speicheradressen auf den Stack auslagern und diese nur bei Bedarf laden und wieder absichern. In Bezug auf Speichereffizienz im Hauptspeicher, benutzen

wir das Minimum  was geht, was die Speicherung einer bitmap Datei angeht. An dieser Stelle haben wir die Aufgabenstellung so interpretiert, dass das  schreiben der einzelnen Pixel innerhalb des Assemblercodes passieren muss. Ist dies nicht der Fall lässt sich der Speicher  um auf $1/3$ des ursprünglich benötigten Speichers verkleinern. Da nun nur noch gespeichert  Werden muss ob der Pixel konvergiert oder nicht. Beim  schreiben des Bildes  C müsste man dann nur noch die Pixelfarben entsprechend dem Eintrag berechnen und die drei Bytes in die Bitmapdatei schreiben.

4.2 Fazit