LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Praktikum Rechnerarchitektur

Gruppe 232 – Abgabe zu Aufgabe A209

Sommersemester 2021

Alara Özdenler Mehmet Emre Özyurt Mateus Messias Mendes

1 Einleitung

Das Konzept der raumfüllenden Kurven begann am Ende des neunzehnten Jahrhunderts und war eine wahre mathematische Kuriosität. Kann eine eindimensionale Kurve eine Fläche oder ein Volumen vollständig bedecken? Dieser Gedanke, war damals sehr neu und die Entdeckung völlig verblüffend. Die Hilbert-Kurve im Besonderen ist zweifellos einer der bekanntesten dieser. Hierbei handelt es sich um eine selbstähnliche und fraktale Raumfüllungskurve, die erstmals von David Hilbert im Jahre 1891 als Verbesserung der Peano-Kurve beschrieben wurde. [1][2]

Der Lauf einer beliebigen Kurve durch den Raum ist oft willkürlich. Dies wäre aber nicht so nützlich wie die Hilbert-Kurve, die im Vergleich dazu garantiert, dass Punkte, die im mehrdimensionalen Raum nahe liegen, auch in linearer Reihenfolge nahe sind. Diese Kohärenz benachbarter Punkte macht man sich zunutze in etlichen Anwendungszwecken wie binäres Suchen, Parallelisierung, Datenbanken, Travelling salesman, etc. [3] [4]. In unserer Arbeit gehen wir dabei in einem kleinen Beispiel auf das Feld des Image Processing ein, bei dem die Hilbert-Kurve bei Image Dithering verwendet wird.

Unser Hauptaugenmerk setzten wir aber vor allem auf die Entwicklung eines iterativen Algorithmus zur Berechnung aller Punkte einer Hilbert-Kurve des Grades n und die darauffolgende Bewertung dieses Ansatzes auf Korrektheit und Performanz. Auf die interessante Frage ob sich ein Punkt der Kurve in konstanter Zeit berechnen lässt, gehen wir ebenso ein.

Was genau ist nun eine Hilbert-Kurve?

Eine Hilbert-Kurve besteht aus 2 Grundelementen: Einheitsquadrate mit einer offenen Seite ($Cups$), und Einheitsvektoren die zwei Cups verbinden ($Joins$). Die rekursive Definition der Kurve fängt bei der Ordnung $n = 1$ an. Diese besteht nur aus einem Cup. Für jede folgende Ordnung besteht eine Hilbert-Kurve der Ordnung n aus 4 Hilbert-Kurven der Ordnung $n - 1$, die in den 4 Quadranten des Raumes platziert sind. Diese

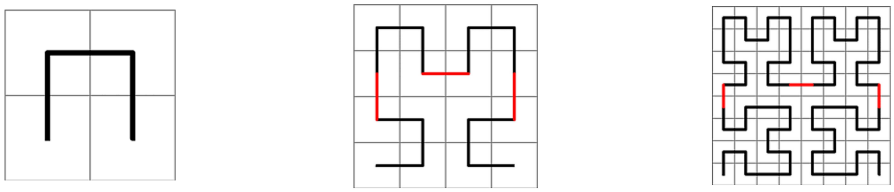


Abbildung 1: Erste 3 Ordnungen der Hilbert-Kurve (2001, Kerry Mitchell)

4 Kurven müssen dabei in geeignete Richtungen gedreht werden, damit sie mit Joins verbunden werden können. Dieser Ansatz wird in der obigen Abb. 1 verdeutlicht. [5] Man beachte, dass, jede Kurve 4^n Punkte besitzt, und aufgrund der Ganzzahligkeit der Koordinaten, diese Punkte alle natürlichen Zahlen im Intervall $[0, 2^n - 1]$ passieren.

2 Lösungsansatz

Unser Algorithmus besteht daraus, für eine Eingabe n eine geordnete Liste an ganzzahligen Koordinaten auszugeben, welche die Abfolge der Knotenpunkte der Hilbert-Kurve n-ten Grades repräsentiert. Wie zuvor erwähnt, soll dieser Ansatz iterativ gelöst werden.

2.1 Iterative Implementierung

Um eine iterative Funktion zu entwickeln, definieren wir zuerst eine Funktion $getPoint()$, die für einen gegebenen Index, d.h. die Position eines Punktes in der Kurve, die x- und y-Koordinaten dieses im Koordinatenraum berechnet und ausgibt. Diese Funktion kann man dann für jeden Punkt der Kurve anwenden, um ihn darauffolgend in der zugehörigen Liste abzuspeichern.

Der wesentliche Bestandteil unseres Algorithmus ist die Ausnutzung der binären Darstellung jedes Indizes. Da es für jeden Cup vier mögliche Positionen gibt, repräsentieren jeweils zwei Bits diese vier Positionen. Die untersten zwei Bits stellen dabei die Knotenposition innerhalb der Hilbert-Kurve des ersten Grades dar. Aus Abb. 1 kann man ablesen, dass die Koordinaten der Kurve $n = 1$ für die Indizes $0, 1, 2, 3$ in dieser Reihenfolge demnach $(0, 0), (1, 0), (1, 1), (0, 1)$ sind. Für größere Ordnungen müssen daraufhin die nachfolgenden zwei Bits von rechts nach links betrachtet werden. [9]

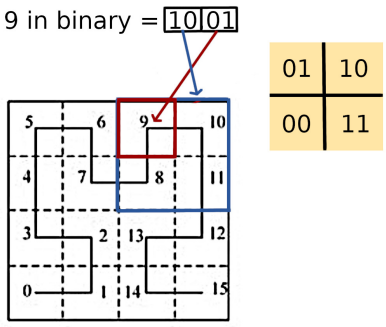


Abbildung 2: Die Position eines Indexes visualisiert durch Bit Darstellung

Diese bestimmen in welchem Quadranten der Kurve des Grades n, die Hilbert-Kurve des Grades $n - 1$ platziert wird und, ob sie davor noch gespiegelt werden muss. Die obere Abb. 2 veranschaulicht dieses Vorgehen.

Um die Koordinaten der Hilbert-Kurve des Grades n zu berechnen, sind also sozusagen die Koordinaten der Hilbert-Kurve des Grades $n - 1$ benötigt. Weil die Koordinaten

für den ersten Grad bekannt sind (siehe oben), können wir somit die Koordinaten für jeden Grad n durch Induktion berechnen. Betrachtet man nun Abb. 3, so erkennt man,

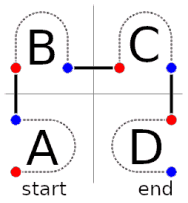


Abbildung 3: Die Positionen der Kurven in 4 Quadranten[6]

dass die Teilbereiche B und C eine einfache Verschiebung der Kurve des Grades $n-1$ darstellen. Die Formel für die Berechnung der Koordinaten dieser Fälle ist unten veranschaulicht, wobei $K(n)$, der Koordinatenvektor (x, y) eines bestimmten Punkts der Kurve n-ten Grades ist und $w = 2^{n-1}$ die Länge der Verschiebung angibt.

$$B : K(n) = K(n-1) + (0, w) \qquad C : K(n) = K(n-1) + (w, w)$$

Der Kurvenbereich A ist eine der obig beschriebenen Spiegelungen der Cups bzw. Kurve des Grades $n-1$ und stellt in diesem Fall eine Spiegelung an der ersten Diagonale/Winkelhalbierenden dar. Somit müssen wir nur die x- und y-Koordinaten der Kurve des vorherigen Grades vertauschen. Sei $K(n).x$ die x- und $K(n).y$ die y-Koordinate des Paares, dann ist:

$$A : K(n).x = K(n-1).y \qquad und \qquad K(n).y = K(n-1).x$$

Der Kurvenabschnitt D stellt eine Spiegelung an der zweiten Diagonale/Winkelhalbierenden dar. Demnach muss man die entgegengesetzten Koordinaten von $w-1$ subtrahieren und am Ende w zu der x-Koordinate addieren, um die Verschiebung nach rechts zu erhalten.

$$D : K(n).x = (w-1) - K(n-1).y + w \qquad und \qquad K(n).y = (w-1) - K(n).x$$

Diese Koordinatenberechnung wird schließlich für jedes 2-Bit-Paar der Bit-Darstellung des Punktes ausgeführt, anfangend bei den untersten Bits. Als Endergebnis erhält man dann die x/y-Koordinaten dieses Punktes im Raum.

2.2 Ausarbeitung in C

Basierend auf dem oben erläuterten iterativen Lösungsansatz haben wir ein C-Programm implementiert, das die x-/y-Koordinaten einer Hilbert-Kurve des Grades n berechnet.

Zunächst implementieren wir die Funktion $getPoint()$, nach dem zuvor beschriebenen Verfahren. Diese kalkuliert demnach für einen gegebenen Index und Grad, die Koordinaten dieses Punktes und gibt diese Werte zurück. Für die Koordinatenwerte definierten wir uns dabei den Datentyp $coord_t$, der ein unsigned Integer Wert ist. Da die

Koordinaten positiv, ganzzahlig und $32Bit$ groß sein sollen, ist dies der bestmögliche Datentyp, um Hilbert-Kurven des höchstmöglichen Grades berechnen zu können.

Wie beschrieben initialisieren wir die lokalen Variablen x und y mit den Koordinaten der Kurve des Grades $n = 1$. Nun benötigt $getPoint()$ eine Schleife, die durch die zwei Bit Paare iteriert und die entsprechende Operation auf die Koordinaten ausführt.

Anstelle einer naiven Schleife, die von $i = 1$ bis zur maximal benötigten Anzahl der 2-Bit Vergleiche dieser Kurve mit i++ inkrementiert, und dann den benötigten Verschiebungswert w für jede Iteration durch 2^i berechnet, erkannten wir direkt, dass wir eine optimierte Schleife erstellen können. Diese wurde direkt im naiven Ansatz und für alle späteren Implementationen verwendet. Die Schliefe beginnt mit $i = 2$, läuft bis $2^{degree}/2$ dieser Kurve, und verdoppelt den Zählindex nach jeder Iteration. Somit kann i direkt als den Verschiebungswert w verwendet werden, wodurch die für den Rechner aufwendigere Berechnung $w = 2^i$ in jeder Iteration entfällt. Die hier beschriebene Anzahl an Iterationen wird aber später im Kapitel zur Berechnungslaufzeit eines Punktes näher erläutert.

Die Hauptfunktion $hilbert()$ ruft $getPoint()$ für alle Punkte der Kurve des Grades $degree$ auf und speichert die erhaltenen $coord_t$-Werte in zwei separaten Listen $xCoordinates$ und $yCoordinates$ des selben Datentyps ab. Die Funktion $printHilbert()$ ermöglicht es, ein Bild dieser Kurve als SVG-Datei zu erstellen und den Lauf der Kurve somit zu visualisieren. Dafür haben wir uns eine vordefinierte SVG-Library für C zunutze gemacht.[8]

Unsere zweite C-Implementation, die wir unter Verwendung der Intrinsics optimierten, basiert auf die Funktionalität der folgenden Ausarbeitung in Assembler, weshalb sie nicht näher beschrieben wird.

2.3 Ausarbeitung in Assembler

Da ein naiver Ansatz in Assembler analog zu dem in C wäre und somit für die weitere Arbeit uninteressant ist, haben wir uns in dieser Ausarbeitung nur auf die optimierte Lösung konzentriert.

Zuerst fanden wir heraus, dass die ganzen callee-saved Register reichen werden für das Programm, was die Verwendung vom Stack deutlich minimierte.

Die wohl wichtigste Optimierung aber ist die Verwendung von SSE Befehlen. Hierbei wurde uns nach dem Entwerfen des Hilbert-Kurven Algorithmus ersichtlich, dass man vier Punkte, die im naiven Ansatz sequenziell berechnet wurden, mit XMM-Registern gleichzeitig berechnen kann. Dies ist darauf zurückzuführen, dass die Punkte beginnend bei der 0 in 4er-Schritten z.B. $0, 1, 2, 3$ oder $12, 13, 14, 15$ in der Binärschreibweise sich nur an den ersten zwei Binärstellen unterscheiden. Da, wie im Algorithmus besprochen, die Iterationsschleife für die Berechnung eines Punktes erst ab der dritten Binärstelle anfängt, hat dies die Implementation mit den XMM-Registern sehr vereinfacht. Hierbei haben wir uns zwei Register ausgesucht, einen für die x-Werte, einen für die y-Werte.

Als Optimierung haben wir diese mit zwei vordefinierten Konstanten initialisiert, die dieselben Startwerte enthielten, wie die Punkte einer 4er-Reihe in der nicht-optimierten Version.

Daraufhin versuchten wir durch geeignete Alignements dieser, die Verwendung von $MOVAPS$ statt $MOVUPS$ Befehlen zu ermöglichen. Hier kamen wir jedoch zu der Realisation, dass dies keinen ersichtlichen Vorteil in der Performanz einbrachte, was auf die modernen Prozessoren zurückzuführen ist, die die unaligned MOV Befehle schon stark optimiert haben. Ohne diese Konstanten hätte man z.B. einige Register benötigt und durch geschickte Shuffle- oder Unpack-Befehle, die x- und y-Koordinaten Startwerte initialisieren können.

Später konnten wir in der Schleife mit geeigneten SSE-Befehlen die Berechnungen für die vier Cases gleichzeitig auf vier Punkten durchführen, was eine erhebliche Optimierung einbrachte. In dem ersten Case ermöglichten uns dabei drei einfache XOR-Befehle eine effiziente Swap-Operation durchzuführen, die wir auch im vierten Case verwendeten.

Eine überraschende Entdeckung machten wir hingegen, als wir uns erhofften, durch das Austauschen der meisten $64Bit$ Register durch $32Bit$ Register eine Optimierung zu erzielen, da die letzteren für die Integer-Werte komplett ausreichen würden. Unser Gedanke hierbei war vor allem, dass diese weniger Speicherplatz im Cache verwenden würden. Die Zeitmessung verriet uns jedoch das Gegenteil, da die $32Bit$ Version ca. 10 Prozent langsamer war. Warum das so ist, können wir nicht genau sagen, unsere Vermutung ist jedoch, dass die 64-Bit x86-Architektur, wie der Name schon sagt für $64Bit$ Register leicht optimiert sein könnte.

Zuletzt haben wir uns noch überlegt, dass eine Sprungtabelle für die vier Cases in der $Switch\text{-}/If\text{-}Else\text{-}Konstruktion$ eine zeitliche Performanz erzielen könnte. Ob dies jedoch schon bei so wenigen Cases beachtlich wäre, wissen wir nicht. Durch die beiden langen Iterationsschleifen, aufgrund der exponentiell wachsenden Punkteanzahl, könnte es aber letztendlich doch sehr sinnvoll sein.

2.4 Laufzeit der Berechnung eines Punktes auf der Hilbert-Kurve

Nehme man an, man möchte einen Punkt $a \in \mathbb{N}_0$ der Hilbert-Kurve des Grades $n \in \mathbb{N}$ berechnen. Nun wäre es interessant zu wissen, ob man diesen in konstanter Zeit berechnen kann. Wir betrachten zwei Fälle:

Hätte man die Koordinaten der Kurve wie in unserer Implementierung schon zuvor berechnet und in der geordneten Reihenfolge oder zumindest in einer Datenstruktur wie eine Hashtabelle abgespeichert, wäre es selbstverständlich möglich, in konstanter Laufzeit $O(1)$ einen beliebigen Punkt abzurufen.

Beispielsweise lässt sich der Punkt a durch einfache indirekte Adressierung mit based mode with displacement and scaling in Assembler erhalten. Sei für $t \in \mathbb{N}_0$ $X(t)$ die x-Koordinate und $Y(t)$ die y-Koordinate von t definiert.

$$X(a) = [\,Startadresse\ der\ xKoordinaten + 4 * a\,] \quad O(1)$$

$$Y(a) = [\,Startadresse\ der\ yKoordinaten + 4 * a\,] \quad O(1)$$

Hat man die Punkte zuvor jedoch noch nicht berechnet, dann ist dieser Lösungsweg nicht sinnvoll, vor allem wenn man nur einige der Punkte der Kurve berechnen will.

In unserem optimierten Algorithmus wird die äußerste Schleife 4^{n-1} mal durchlaufen, wobei die innere Schleife zur Berechnung der Punkte $n-1$ mal durchlaufen wird, wodurch die Berechnung aller Punkte in $O(n * 4^n)$ liegt. Des Weiteren sollte man den Speicherplatzbedarf in Betracht ziehen, der mit $2 * 4^n$ Koordinatenwerten exponentiell wächst. Deshalb konzentrieren wir uns nun auf den zweiten Fall.

Schaut man sich die Definition der Hilbert-Kurve an, wird klar, dass die Berechnung eines einzelnen Punktes ohne jegliche Vorberechnungen oder Speicherung von Werten in konstanter Zeit, das heißt unabhängig des gewählten Grades, unmöglich ist, da diese rekursiv bestimmt ist. Nehmen wir jedoch an, man hätte nur den Grad n der Hilbert-Kurve und suchte den Punkt $a \, \epsilon \, \mathbb{N}$. Hier wird der erste Punkt an der Stelle $a = 0$ weggelassen, da dieser konstant an der Stelle $(0|0)$ liegt. Wie schon zuvor ausführlich beschrieben, benötigt man in unserem Algorithmus für die Berechnung eines Punktes dessen Bit-Darstellung. Diese Anzahl der benötigten Bits $B(t)$; $t \, \epsilon \, \mathbb{N}$ für den Punkt a

$$B(a) = \lfloor log_2(a) \rfloor + 1$$

kann halbiert werden, da wir immer 2 Bits gemeinsam betrachten in jeder Iteration. Da zudem die ersten beiden Bits immer dieselben sind für alle Punkte, unabhängig vom Index oder Grad, benötigt man eine Iteration weniger. Dadurch erhält man für die Anzahl der Iterationen $I(t)$; $t \, \epsilon \, \mathbb{N}$ für a:

$$I(a) = \left\lceil \frac{\lfloor log_2(a) \rfloor + 1}{2} \right\rceil - 1$$

Da es für Grad n eine Anzahl von 4^n verschiedene Punkten gibt, muss man jedoch den Worst Case betrachten, die größtmögliche Zahl 4^{n-1}. Wenn man diese Zahl in die obige Formel für die Berechnung der Anzahl der Iterationen einsetzt, erhält man die maximale Anzahl an Iterationen $I(4^{n-1}) = n - 1$.

Nun gibt es den Fall, dass die Zahl a in den oberen Bits nur aus Nullen besteht. In diesem Fall muss man bei diesen restlichen Iterationen nur die x- und y-Koordinate m-mal vertauschen. Um zu bestimmen, ob dies überhaupt nötig ist, subtrahiert man die Anzahl an Iterationen für a von der zuvor berechneten maximalen Anzahl an Iterationen.

$$m = I(4^{n-1}) - I(a) = n - 1 - I(a)$$

Ist m ungerade, müssen die x- und y-Koordinate vertauscht werden, ist der Wert gerade, bleiben die Koordinaten erhalten. Die Bestimmung von m und der eventuelle Tausch der Koordinaten liegt in $O(1)$.

Innerhalb der Schleife benötigen die einzelnen Berechnungen für die 4 verschiedenen Fälle ebenfalls nur $O(1)$ Laufzeit. Zusammenfassend kann man sagen, dass der Best Case in $O(1)$ liegt, wenn $a = 0$ bzw. sehr klein ist. Der Worst Case benötigt, wie zuvor berechnet, $n - 1$ Iterationen und liegt dadurch in $O(n)$. Geht man für den Average Case $A(n)$ der Kurve mit Grad $n \, \epsilon \, \mathbb{N}$ davon aus, dass die Punkte zufällig für eine Suche gewählt werden, kommt man zu dem Schluss, dass im Durchschnitt

$$A(n) = \begin{cases} 0 & x = 1 \\ \sum_{i=1}^{n-1} \frac{3i}{4^{n-i}} = \frac{4}{3}(4^{-n} - 1) + n & x > 1 \end{cases}$$

Schleifendurchläufe nötig sind. Wenn n gegen Unendlich geht, sind also $n - \frac{4}{3}$ Iterationen nötig, wodurch die Laufzeit auch im Average Case in $O(n)$ liegt. Schlussendlich ist die Laufzeit der bestmöglichen Punkt-Berechnung in unserem Algorithmus abhängig vom Grad der Kurve und somit nicht konstant sondern linear.

Einen erheblichen Vorteil erhält man durch diesen Algorithmus jedoch trotzdem, wenn man in Betracht zieht, dass mit steigendem Grad die Berechnung eines einzelnen Punktes nur linear in $O(n)$ wächst, mit steigendem Grad die Anzahl an Punkten jedoch exponentiell wächst, und zwar in $O(4^n)$.

3 Korrektheit

In den vorherigen Kapiteln wurde die Vorgehensweise der Punktberechnung schon ausführlich erläutert. Da der komplette mathematische Beweis den Rahmen dieser Arbeit sprengen würde, haben wir uns dafür entschieden, die Korrektheit anderweitig zu beweisen.

Zuerst sollte man sich im Klaren sein, welche Grenzen diese Implementierung beinhaltet. Einerseits ist die Hilbert-Kurve nur für ganzzahlige Grade $n > 0$ definiert, weshalb nur positive Ganzzahlen als Eingabewerte für $degree$ in Frage kommen. Andererseits wurde im Lösungsansatz schon erwähnt, dass unsere Konzentration auf $32Bit$ Integer Werten für die Speicherung der Punkte liegt, genauer unsigned Integer Zahlen. Diese können also im Intervall $[0; 2^{32} - 1]$ liegen. Zuletzt müssen wir aber beachten, dass der Parameter size bei der dynamischen Speicherallokierung $malloc(size_t\ size)$ den Maximalwert $2^{64} - 1$ in einer $64Bit$-Architektur besitzt. Für unsere Allokation $malloc(points * sizeof(coord_t))$ muss also $points * sizeof(coord_t) < 2^{64} - 1$ gelten. Da die Anzahl an Punkten 4^n ist, und $coord_t$ ein $32Bit$ unsigned Integer ist, erhalten wir die Ungleichung $4^n * 4 < 2^{64} - 1$. Durch Lösen dieser Ungleichung ist klar, dass unsere Implementierung nur Kurven bis zum maximalen Grad $n = 30$ korrekt berechnet.

Diese Grenzen unserer Implementierung erlauben es nun, die wenigen Fälle für die Grade n im Intervall $[1; 30]$ empirisch auf die Korrektheit zu beweisen.

Einerseits ermöglicht es unser Rahmenprogramm, die Hilbert-Kurve visuell zu überprüfen. Wichtig ist es darauf zu achten, dass keine Überschneidungen oder ähnliches auftreten. Hierfür haben wir das Argument $-p$ integriert. Bei großen n wird aber ersichtlich, dass zum einen die visuelle Überprüfung sehr mühsam wird und zum anderen ab einem Grad von n = 11 unmöglich ist, da die Kurve mehr Punkte besitzt als die Anzahl an Pixel eines Full-HD Bildschirms.

Deshalb haben wir die Methode $testCorrectness()$ im Rahmenprogramm eingebaut, die die Punkte automatisch mit unserer ersten Referenzimplementierung vergleicht. Hierfür mussten wir zu Beginn unseres Projekts die Methode $firstTest()$ konstruieren, wo unsere Referenzmethode $getPoint()$ mit der Methode $hilbertEps(t, eps)$ nebeneinander ausgeführt und Punkt für Punkt verglichen. Letztere basiert dabei auf den Pseudocode von Michael Bader[1] zur Berechnung eines Hilbert-Punktes. Es wurde gezeigt, dass für jedes $n <= 18$, beide Implementierungen in allen Punkten übereinstimmen. Für größere n wurde dabei aufgrund der sehr hohen Anzahl an Punkten aus praktischer und

zeitlicher Sicht auf die weitere Testung verzichtet. Weshalb die Überprüfung für größere n jedoch relativ vernachlässigbar ist, wird im Abschluss dieses Kapitels ersichtlich. Demnach konnte für die folgende Ausarbeitung die Korrektheit unseres ersten Ansatzes bis $n = 18$ angenommen werden.

Im weiteren Verlauf des Projekts wurde dann die zuvor erwähnte Methode $testCorrectness()$ entwickelt, die mit dem Argument $-tc$ ausgeführt werden kann. Hierbei werden die gespeicherten Punkte der Assembler Implementierung mit den Punkten, die durch die $getPoint()$ kalkuliert werden, auf Richtigkeit verglichen. Nach Ausführen des obigen Befehls wird in der Konsole angezeigt, ob alle Punkte korrekt berechnet wurden. Wurde ein falscher Punkt berechnet, terminiert die Methode mit einem Hinweis, welcher Punkt fehlerhaft ist. Bei unseren Tests wurden alle aus praktischer Sicht erlaubten Hilbert-Kurven als richtig anerkannt.

Es ist noch darauf hinzuweisen, dass diese Überprüfungsmethode von dem gewählten zentralen Rechner, dessen Betriebssystemimplementierung und physischen Speicherplatz abhängig ist, da für die Allokation großer Grade eine hohe Speicherkapazität benötigt wird. Beispielsweise werden $34.36GB$ für $n = 16$ und $137.43GB$ für $n = 17$ benötigt. Dadurch scheitert die Allokation mit $malloc()$ bei kommerziellen Computern meist schon bei $n = 15$. Um die Grade 16 und 17 zu überprüfen, musste hier deshalb auf die Rechnerhalle zugegriffen werden.

4 Performanzanalyse

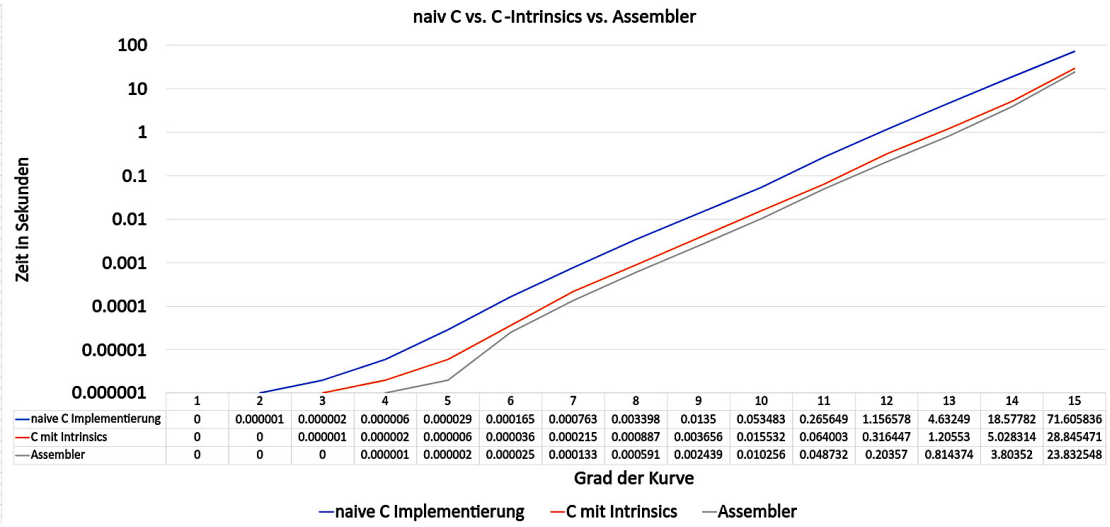
Die Analyse der Performanz eines Algorithmus ist zweifellos ein wichtiger Aspekt, den man nach der Entwicklung betrachten sollte. Um diese zu analysieren, haben wir die naive C Implementierung, die C Implementierung mit Intrinsics und unsere optimierte Haupt-Assembler-Implementation verglichen.

Um die Performanz zu berechnen, haben wir immer die durchschnittliche Zeit für jede Ordnung betrachtet. Für den Mittelwert führten wir für die Grade 1 bis 7 insgesamt 1.000.000, für die Grade 8 bis 11 total 100.000 und für die letzten Grade bis 15 im Ganzen 1000 Iterationen aus. Hiermit wurde versucht, die leistungsmindernden Effekte anderer Prozesse im Hintergrund zu minimieren, um ein möglichst genaues Ergebnis zu erhalten. Die Laufzeitmessung erfolgte auf einer Intel Core i5-8265U CPU mit 8 Kernen, 1,60 GHz, 8 GB Arbeitsspeicher, Ubuntu 20.10 mit Linux Kernel 5.8.0. Alle Versionen wurden mit GCC 9.3.0 mit der Optimierungsoption -O1 kompiliert.

Idealerweise kann unser Algorithmus wie im oberen Kapitel beschrieben maximal Kurven der 30. Ordnung erzeugen. Die Testumgebung mit 8 GB Arbeitsspeicher kann aber aufgrund des 34 GB Speicherbedarfs für die 16. Ordnung, maximal die 15. Ordnung erzeugen und dies auch nur unter Zuhilfenahme von swapping, was die Performanz in diesem Fall natürlich etwas verschlechtert.

Die nachfolgende Grafik vergleicht die Leistung aller unserer Implementierungen. Aufgrund der großen Zeitdifferenz zwischen den Ordnungen ist es sehr schwierig, die prozentualen Unterschiede zu visualisieren. Daher verwendet die y-Achse, die der Zeit in Sekunden entspricht, die logarithmische Skala von 10^{-6} bis 10^{2}. Die x-Achse zeigt die

Ordnung der Kurven.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
naive C Implementierung	0	0.000001	0.000002	0.000006	0.000029	0.000165	0.000763	0.003398	0.0135	0.053483	0.265649	1.156578	4.63249	18.57782	71.605836
C mit Intrinsics	0	0	0.000001	0.000002	0.000006	0.000036	0.000215	0.000887	0.003656	0.015532	0.064003	0.316447	1.20553	5.028314	28.845471
Assembler	0	0	0	0.000001	0.000002	0.000025	0.000133	0.000591	0.002439	0.010256	0.048732	0.20357	0.814374	3.80352	23.832548

Abbildung 4: Graphischer Vergleich der unterschiedlichen Implementationen

Der Leistungsunterschied zwischen dem naiven C und Assembler ist sehr einfach zu erkennen. Die C-Implementierung mit einer naiven for-Schleife ist um einiges schlechter als die Assembler-Version. Dies ist das Ergebnis der SSE-Optimierungen, die es uns erlaubten, vier x- und vier y-Koordinaten gleichzeitig zu berechnen. Das führt natürlich dazu, dass die Assembler-Laufzeit durchschnittlich 3-4-mal schneller ist als die C-Version. Dieser Unterschied ist bei jeder Ordnung fast gleich aufgrund des linearen Verhaltens beider Algorithmen , was auch im Graphen durch das lineare Steigungsverhalten beider Linien visualisiert erkennbar ist.

Wie in Abb. 4 zu sehen, gibt es im Gegensatz zu dem vorherigen Fall keinen erheblichen Unterschied zwischen der C-Implementierung mit Intrinsics und der Assembler-Implementierung. Die Assembler-Implementierung ist ab der sechsten Ordnung durchschnittlich 1,5-mal schneller als C mit Intrinsics. Dies liegt daran, dass der durch Intrinsics generierte Assembler-Code hauptsächlich die gleichen Befehle wie unsere Implementierung benutzt. Es handelt sich jedoch immer noch nicht um reinen Assembler-Code, was bedeutet, dass der Compiler während der Kompilierung zusätzliche unnötige Befehle hinzufügen kann, wodurch dieser Leistungsunterschied entsteht. Es ist noch zu beachten, dass der späte Kurven-Start nur die Folge von sehr kleinen Zeit-Werten ist, die nicht auf dieser Graphik dargestellt werden können.

5 Anwendung der Hilbert-Kurve im Feld des Image Processing

Als Beispiel betrachten wir ein digitales Halbtonverfahren. Bei einem Halbtonverfahren wird ein Graustufenbild durch eine Verteilung von schwarzen Punkten auf weißem

Hintergrund approximiert. Das menschliche Auge kann diskrete Punkte nicht exakt wahrnehmen, da wir in unserem Sichtfeld nur die durchschnittlichen Intensitäten sehen, die kleinen Raumwinkeln entsprechen. Dadurch entsteht die Illusion eines vollständigen Graustufenbildes. Digitales Halbtonverfahren, auch bekannt als räumliches Dithering, bedeutet mithilfe von Computeralgorithmen zu entscheiden, wie diese Punkte platziert werden.

Wir betrachten die Darstellung Graustufenbilder in Bilevel-Anzeigen. Eine Bilevel-Anzeige ist ein Bildschirm, der eine einzige Farbe auf einen Hintergrund anzeigt (in unserem Fall schwarz auf weiß). Durch Dithering werden Pixel effektiv so verteilt, dass die durchschnittliche Farbintensitäten in kleinen Regionen des geditherten Bildes ungefähr die von dem ursprünglichen Graustufenbild entsprechen. Damit liegt der durchschnittliche Quantisierungsfehler nahe Null. Der Quantisierungsfehler von einer Region im Bild entspricht der Differenz zwischen der Summe der Farbintensitäten in einer Region im Originalbild, und der Summe von derselben Region im geditherten Bild. [7]

Zuvor existierende übliche Dithering-Algorithmen wurden für eine bestimmte Klasse von Grafikgeräten entwickelt und funktionierten nicht auf hochauflösenden Hardcopy-Geräten wie Laserdruckern. Daher wurde eine digitale Halbtontechnik eingeführt, die Hilbert-Kurven verwendet und in einer Vielzahl von Geräten verwendet werden kann.

Das digitale Halbtonverfahren besteht aus den Schritten der Unterteilung des Quellbildes in kleinen Regionen basierend auf dem Verlauf der Hilbert-Kurve, der Berechnung der durchschnittlichen Intensitäten jeder Region und der Bestimmung der Punktmuster des geditherten Bildes entsprechend jeder Intensität.

Dieses Verfahren verwendet den Verlauf der Hilbert-Kurve, um den Quantisierungsfehler über das Bild gleichmäßig zu verteilen. Mit einer Hilbert-Kurve ist es möglich, alle Pixel auf einem Rasterbild eindeutig anzusprechen. Es ermöglicht, eine Unterteilung des Bildes in Regionen, wobei jede Region auf ein Teilintervall des Einheitsintervalls abgebildet wird. Dies bietet die Möglichkeit, alle Unterregionen eines Bildes zu besuchen, und demzufolge jeden Punkt in jeder Unterregion zu besuchen. Somit stellt es ein effektives Verfahren dar, auf ein Rasterbild Dithering auszuführen. Dies führt zu einem Bild, das im Gegensatz zu traditionellen Rastermustern, frei von sichtbaren horizontalen Scanlinien ist. Dadurch wird der Gittereffekt minimiert, der sich häufig bei anderen Dithering-Verfahren manifestiert, z.B. die Standardverfahren der Bildabtastung.

In Abb. 5 sieht man zwei Versionen eines Bildes, das Original und das mit dieser Methode gedithert wurde. Der Hilbert-Kurven-Dithering-Algorithmus erzeugt aperiodische Muster von gleichmäßig verteilten Punkten ohne horizontale Scanlinien. Es gibt die Graustufen gut wieder und erfasst die feinen Details. Diese Merkmale sind in Abb. 5 erkennbar, insbesondere in Gesicht, Augen und Haaren des Jungen.

Die Hilbert-Kurvenmethode hat gegenüber den älteren Ditheringmethoden mehrere Vorteile. Die erzeugten Muster sind wahrnehmbar angenehm mit ähnlichen Eigenschaften wie die fotografische Kornstruktur. Der Algorithmus ist rechnerisch effizient und erfordert nur eine Addition, eine Subtraktion und ein Vergleich pro verarbeitetem Bildelement. [7]

(a) Original

(b) Hilbert-Kurven-Dithering

Abbildung 5: Vergleich des Originalbilds mit dem geditherten Bild[7]

6 Zusammenfassung und Ausblick

In unserer Arbeit hatten wir das Ziel, dem Leser das Prinzip der raumfüllenden Hilbert-Kurve näher zu bringen. Von den zahlreichen Anwendungszwecken raumfüllender Kurven gaben wir das nennenswerte Anwendungsbeispiel des Dithering im Feld des Image Processing wieder. Unser Fokus lag auf die Erarbeitung eines Algorithmus für die Berechnung einer Hilbert-Kurve und die spätere Analyse auf Korrektheit und Performanz.

Unseres Erachtens erarbeiteten wir eine sehr effiziente Implementation der Hilbert-Kurve, die einerseits nur auf iterative Lösungsansätze beruht statt performant schlechtere und für Implementationen in der Literatur übliche rekursive. Zum anderen erzielten wir noch zusätzlich, unter Verwendung der SSE-Erweiterungen und weiterer kleiner Optimierungen, eine annähernd vierfache Beschleunigung.

Die Grenzen unserer Implementierung haben wir dabei deutlich hervorgehoben. Diese sind jedoch durch etwas mehr Zeit und Handwerk möglicherweise ausbaufähig.

Für Prozessoren mit der ISA-Erweiterung AVX könnte man zum Beispiel durch Verwendung der YMM-Register, die Berechnung der Punkte von Kurven der Ordnung $n > 1$ theoretisch noch weiter beschleunigen. Hier könnte man zum Beispiel mit vier Registern 16 Punkte gleichzeitig betrachten.

Eine weitere mögliche Optimierung, die es erlauben würde, die Vielzahl an Punkten von Kurven größerer Ordnungen abzuspeichern, wäre die zusätzliche Verwendung einer Festplatte statt der ausschließlichen Nutzung des Random Access Memorys. Hierbei muss man sich aber natürlich bewusst sein, dass dies einen weitere erhebliche Verlangsamung mit sich bringen würde, zu der ohnehin schon breiten Menge an Berechnungen.

Ob schlussendlich Hilbert-Kurven so hoher Ordnung in naher Zukunft schon von Nutzen sein werden, lässt sich allerdings nur spekulieren.

Literatur

[1] Michael Bader. *Space-Filling Curves - An Introduction With Applications in Scientific Computing*. Springer Science and Business Media, Berlin Heidelberg, 2012.

[2] David Hilbert. Über die stetige abbildung einer linie auf ein flächenstück. In *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes*, pages 1–2. Springer, 1935.

[3] P. H. J. King J. K. Lawder. Querying Multidimensional Data Indexed Using the Hilbert Space Filling Curve. *Birkbeck College, University of London*, 2001. `https://sigmodrecord.org/publications/sigmodRecord/0103/3.lawder.pdf`, visited 2021-06-28.

[4] Peter Oberhofer. Anwendung raumfüllender Kurven und Parallelisierung. *Technische Universität München, Institut für Informatik*, pages 10–15, 2003. `https://www5.in.tum.de/lehre/seminare/oktal/SS03/ausarbeitungen/oberhofer.pdf`, visited 2021-06-24.

[5] Nicholas J Rose. Hilbert-type space-filling curves. *North Carolina State University, Chicago, IL*, 2001.

[6] Problem statement for hilbert. (accessed 2021-07-07) `https://community.topcoder.com/stat?c=problem_statement&pm=2376`.

[7] Luiz Velho and Jonas de Miranda Gomes. Digital halftoning with space filling curves. *ACM SIGGRAPH Computer Graphics*, 25(4):81–90, 1991. `https://www.visgraf.impa.br/Courses/npr07/materials/0%20intro/4%20points/p8-velho.pdf`, visited 2021-06-24.

[8] Chris Webb. Svg C-Library. *CodeDrome*, 2018. `https://www.codedrome.com/svg-library-in-c/`, visited 2021-05-24.

[9] Edmund Weitz. *Elementare Differential Geometrie (nicht nur) für Informatiker*. Springer-Verlag, 2019.