LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Praktikum Rechnerarchitektur

Gruppe 142 – Abgabe zu Aufgabe A213

Sommersemester 2020

Anton Baumann Felix Brandis Michal Cizevskij

1 Einleitung

Im Rahmen des Praktikums Rechnerarchitektur hat sich unsere Gruppe mit dem Thema der Moore-Kurve auseinandergesetzt. Die Moore-Kurve ist eine von dem Mathematiker Eliakim Hastings Moore in 1900 entwickelte raumfüllende Kurve. Sie basiert auf der Hilbert-Kurve und findet oft Anwendung in Bildverarbeitung und Datendarstellung. Unsere Aufgabe war es, einen Algorithmus in Assembler zu entwickeln, der die Koordinaten einer Moore-Kurve beliebigen Grades berechnet. Hierbei sollten wir auch beantworten, ob es möglich ist, einen beliebigen Punkt der Kurve in konstanter Zeit unabhängig vom Grad zu berechnen.

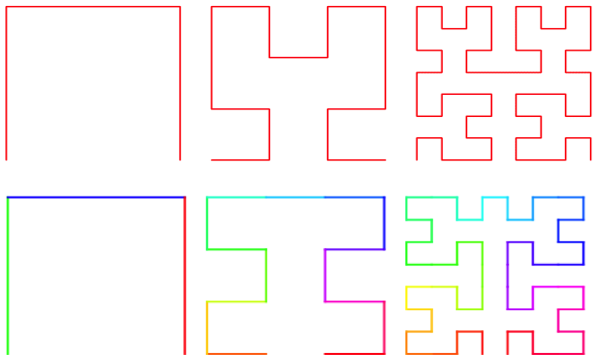


Abbildung 1: Die drei ersten Iterationen der Hilbert- und Moore-Kurve [1], [4]

1.1 Charakterisierung der Moore-Kurve und weiteren raumfüllenden Kurven

Raumfüllende Kurven sind Linien, die jeden Punkt des Intervalls $[0,1]^d$ durchlaufen. Diese können im zweidimensionalen Raum als surjektive Abbildungen dargestellt werden, welche vom Einheitsintervall $[0,1]$ in das Einheitsquadrat $[0,1]^2$ abbilden.
Eine wichtige Unterkategorie der raumfüllenden Kurven ist FASS („space-filling, self-avoiding, simple and self-similar“). Solche Kurven erfüllen viele oft in der Praxis geforderte Bedingungen: Zu einem ist jede FASS-Kurve selbst ausweichend. Das bedeutet, dass jeder Punkt exakt einmal durchlaufen werden muss. Außerdem behalten die FASS-Kurven die Lokalität der Punkte bei. Sind Punkte räumlich nah im eindimensionalen Einheitsintervall, so gilt dies ebenfalls für ihr Bild in $[0,1]^d$ (siehe Abb. 2).

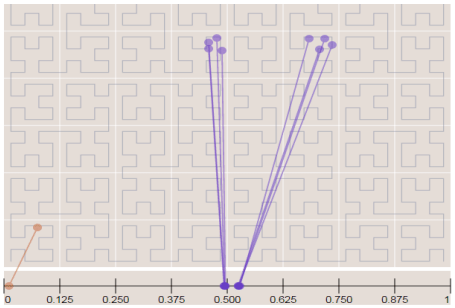


Abbildung 2: Lokalität einer raumfüllenden Kurve [8]

Eine sehr bekannte Kurve, die die FASS-Eigenschaften erfüllt, ist die Hilbert-Kurve. Diese wurde von dem gleichnamigen Mathematiker David Hilbert in 1891 entwickelt und findet aufgrund ihres einfachen Aufbaus und ihrer Stetigkeit besonders oft in der Praxis Verwendung.
Die Moore-Kurve ist eine Variation der Hilbert-Kurve, welche sich aus vier rotierten Hilbert-Kurven geringeren Grades zusammensetzt. Der wesentliche Unterschied zwischen den beiden Kurven ist, dass die Moore-Kurve im Gegensatz zur Hilbert-Kurve eine geschlossene Form hat, Anfangs- und Endpunkt also benachbart sind.
In raumbezogenen Anwendungen kommt es oft vor, dass raumfüllende Kurven verwendet werden, um effizient geographische Daten, die in zwei- oder dreidimensionaler Form vorhanden sind, auf eindimensionale Indizes zu abzubilden. Hierfür wird das Gebiet in Zellen aufgeteilt, welche zu einer kompakten Darstellung der bestimmten Region dienen. Dabei ist es erwünscht, dass geographische Nähe auch in der Abbildung auf den eindimensionalen Raum erhalten bleibt. Benachbarte Bilddaten liegen somit nebeneinander im Speicher und können daher effizienter geladen werden.

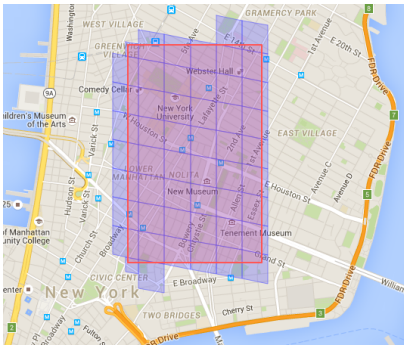


Abbildung 3: Einteilung von New York in Zellen [6]

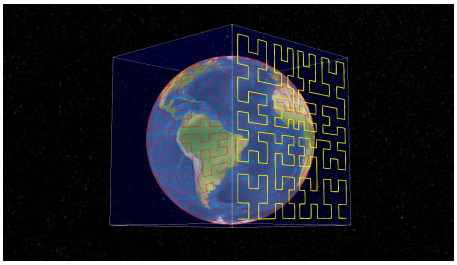


Abbildung 4: Hilbert-Kurve projiziert auf die Erde [6]

Da eine FASS-Kurve genau diese Anforderungen erfüllt, eignet sie sich besonders gut für solch eine Verwendung. Ein konkretes Beispiel der Anwendung wäre die S2-geometry-library von Google.

1.2 Benutzerdokumentation

Der Quellcode unseres Programms enthält ein Makefile. Somit kann auf einem Computer, der *AVX2* unterstützt der Code durch einen Aufruf von `make` kompiliert werden. Auf älteren Maschinen kann mit `make disable_avx` gebuildet werden.
Um eine Moore-Kurve *6. Grades* mithilfe der Assembler-Implementierung zu berechnen und unter *moore.svg* abzuspeichern kann man nun die Binärdatei mit `./moore --implementation asm --degree 6 --path moore.svg` ausführen. Um die verschiedenen Implementierungen zu benchmarken kann das Programm wie folgt im Benchmark-Modus aufgerufen werden: `./moore --benchmark --degree 6 --repetitions 1000 --write_results`. Da hier die *write results* Flag gesetzt ist, wird für jede getestete Implementierung das Ergebnis der letzten Iteration als *.svg* Datei gespeichert.

2 Lösungsansatz

2.1 Die Moore-Kurve

Wie im letzten Abschnitt erwähnt, kann die Moore-Kurve des Grads n aus vier Kopien der Hilbert-Kurve des Grads $n-1$ erzeugt werden (für $n > 1$), siehe dazu auch Abbildung 5. Das Problem kann also heruntergebrochen werden auf die Erzeugung der Hilbert-Kurve und dem anschließendenden Kopieren, Verschieben und Rotieren der Punkte.
Es gibt rekursive Algorithmen für die Erzeugung der Hilbert-Kurve, diese klammern wir im Folgenden jedoch aus. (Siehe Aufgabenstellung)

2.2 Erzeugung der Hilbert-Kurve: iterativer Punkt-für-Punkt Ansatz

Im Folgenden soll ein Algorithmus dargestellt werden, der für gegebenen Index i und Grad n die Koordinaten des korrespondierenden Punktes an der Stelle i der Moore-Kurve berechnet. Geht man davon aus, schon einen solchen funktionierenden Algorithmus für die Hilbert-Kurve zu haben, so muss nur der Quadrant des Punktes auf der Moore-Kurve bestimmt werden, dann der entsprechende Punkt der Hilbert-Kurve errechnet und je nach Quadrant verschoben, gespiegelt oder rotiert werden. Dies wird in Abbildung 5 grafisch veranschaulicht und im Algorithmus 1 (alle Algorithmen im Anhang) skizziert. Damit Algorithmus 1, der einen erzeugten Hilbert-Punkt von Grad

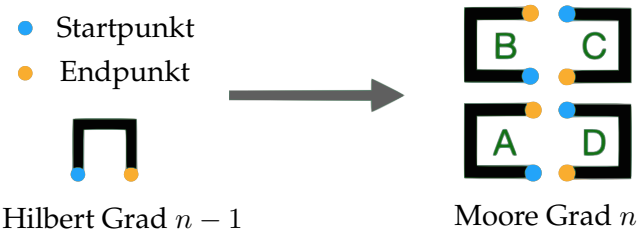


Abbildung 5: Transformation von Hilbert-Kurve des Grads n-1 zu Moore-Kurve des Grads n

$n-1$ nur noch verschiebt, funktioniert, fehlt noch eine Methode, die für gegebenen Index i und Grad n die Koordinaten des entsprechenden Punktes auf der Hilbert-Kurve zurückgibt.
Abbildung 6 zeigt, wie das funktioniert: Für jeden Punkt beginnt man mit den Koordinaten $(0, 0)$. Wird der Index in Binärdarstellung umgewandelt, so können mit den least significant Bits beginnend immer zwei Stellen angesehen werden und dann nach dem in der Graphik dargestellten Entscheidungsmuster die Koordinaten durch Spiegelung und Verschiebung manipuliert werden. Durch iterative Ausführung und mit jeder Iteration einer Vergrößerung des Offsets können so die Koordinaten jedes Punktes bestimmt werden. [3] Algorithmus 2 zeigt eine mögliche Formalisierung.

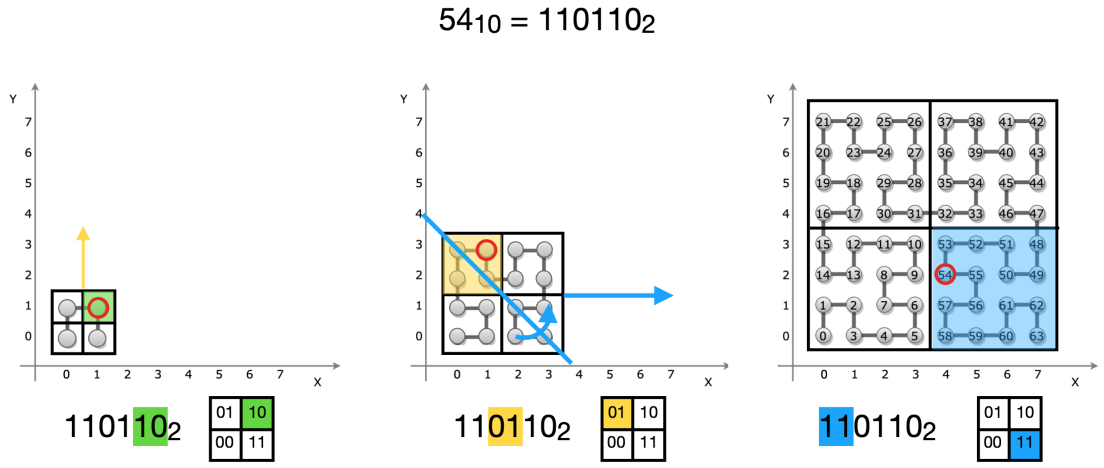


Abbildung 6: Bottom-up Berechnung der Koordinaten eines Punkts der Hilbert-Kurve, Beispiel für Punkt mit Index 54 und Hilbert-Kurve des Grads 3 (Adaptiert von [3])

2.3 Erzeugung der Hilbert-Kurve: Dynamischer Ansatz

Der oben dargestellte Algorithmus hat den Vorteil, für einen gegebenen Index und Grad die Koordinaten des korrespondierenden Punktes ausrechnen zu können, ohne dafür die gesamte Kurve erzeugen zu müssen. Soll allerdings die gesamte Kurve erstellt werden, also x- und y-Werte für alle Punkte errechnet werden, so ist der iterative Aufruf des oben beschriebenen Algorithmus für jeden Index sehr ineffizient.
Ebenso wie die Moore-Kurve des Grads n aus vier Kopien der Hilbert-Kurve des Grads $n-1$ aufgebaut werden kann, kann auch die Hilbert-Kurve des Grads n aus vier Kopien der Hilbert-Kurve des Grads $n-1$ erzeugt werden (Abbildung 7). Man kann also bei der Erzeugung der Hilbert-Kurve ähnlich vorgehen wie beim Algorithmus 1, der aus der Hilbert-Kurve des Grads $n-1$ die Moore-Kurve des Grads n erzeugt: Mit gegebenen Punkten der Hilbert-Kurve des Grads $n-1$ lassen sich alle Punkte der Hilbert-Kurve des Grads n erzeugen. Die Spiegelungen, Rotationen und Translationen für die jeweiligen Quadranten sind natürlich anders als in Algorithmus 1.

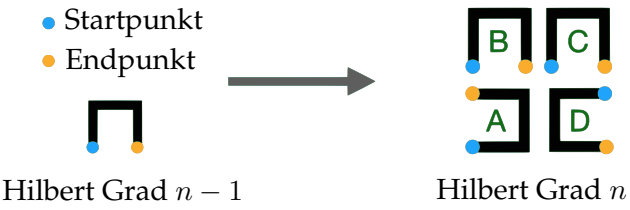


Abbildung 7: Transformation von Hilbert-Kurve des Grads $n-1$ zu Hilbert-Kurve des Grads n

Dieser Ansatz kann zwar keinen einzelnen Punkt an einem gegebenen Index liefern ohne die gesamte Kurve zu berechnen, die Berechnung aller Punkte der Kurve ist dafür aber bei weitem effizienter als der iterative Aufruf des Punkt-zu-Punkt Ansatzes, da Ergebnisse schon durchgeführter Berechnungen wiederverwertet werden. Algorithmus 3 zeigt, wie zunächst die Hilbert-Kurve bis Grad $n-1$ aufgebaut und daraus dann die Moore-Kurve des Grads n erzeugt wird. Für die Nachvollziehbarkeit sind Abbildung 5 und 7 essentiell.

2.4 Theoretische Laufzeitanalyse und Vergleich

Man kann sehen, dass der Aufwand für die Berechnung eines einzelnen Punktes beim Punkt-für-Punkt Algorithmus (Algorithmus 2) vom Grad der Kurve abhängt: In der Unterfunktion $berechneHilbertKoordinate(index, degree)$ wird eine Schleife $log_2(2^{degree})-1 = degree-1$ mal durchlaufen. Da der Aufwand der Operationen innerhalb der Schleife konstant ist, steigt der Rechenaufwand pro Punkt in Algorithmus 2 also linear mit dem Grad der Kurve.
Für Algorithmus 3 hingegen lässt sich zeigen, dass der durchschnittliche Aufwand für die Berechnung eines Punktes konstant ist, unabhängig vom Grad der Kurve.
Algorithmus 3 besteht aus dem einmaligen Aufruf der Hilbert-Unterfunktion und dann dem Verschieben der erhaltenen Punkte zur Erzeugung der Moore-Kurve.
In der Hilbert-Unterfunktion gibt es zwei ineinander verschachtelte Schleifen: eine, die die Indexvariable j von 2 bis inklusiv $degree-1$ durchläuft (Denn die Hilbert-Unterfunktion wird ja mit $degree-1$ aufgerufen) und die innere Schleife, bei der die Punkte des letzten Grades kopiert und verschoben werden.
Diese innere Schleife wird $quarter = 4^{j-1}$ mal durchlaufen, da immer alle Punkte der Kurve des Grads $j-1$ transformiert werden müssen, und sie hat konstanten Aufwand c. In ihr wird ein Punkt aus dem Speicher geladen, die Koordinaten werden manipuliert und an 4 Stellen wieder in den Speicher geschrieben. (Die Annahme der Konstanz ist hier natürlich theoretisch: sie berücksichtigt keine Zeitunterschiede durch Caching oder RAM-Limits, sondern geht von einem idealisierten Maschinenmodell ohne die Probleme des Von-Neumann-Bottlenecks aus)
Nach Durchlaufen der Hilbert-Unterfunktion werden in der Moore-Funktion dann noch einmal $4^{degree-1}$ Punkte kopiert und an die richtige Stelle geschoben.
Der Aufwand für die Berechnung aller Punkte der Moore-Kurve des Grads n $Op(n)$

kann also folgendermaßen rekursiv definiert werden:

$$\begin{aligned} I &: Op(n) = Op(n-1) + 4^{n-1} \cdot c \\ II &: Op(1) = c \end{aligned}$$

Diese rekursive Gleichung kann nun aufgelöst werden:

$$\begin{aligned} Op(n) &= Op(n-1) + 4^{n-1} \cdot c \\ &= Op(n-2) + 4^{n-2} \cdot c + 4^{n-1} \cdot c \\ &= c \cdot (4^0 + ... + 4^{n-2} + 4^{n-1}) \\ &= c \cdot \sum_{i=0}^{n-1} 4^i \\ &= c \cdot \frac{1}{3}(4^n - 1) \end{aligned}$$

Eine Kurve von Grad n hat Seitenlänge 2^n und somit 4^n Punkte. Der durchschnittliche Aufwand pro Punkt $avg\_Cost(n)$ lässt sich wie folgt berechnen:

$$avg\_Cost(n) = \frac{Op(n)}{4^n} = c \cdot \left(\frac{4^n}{3 \cdot 4^n} - \frac{1}{3 \cdot 4^n} \right) = \frac{1}{3}c - c \cdot \frac{1}{3 \cdot 4^n}$$

$$\Rightarrow \lim_{n \to \infty} avg\_Cost(n) = \lim_{n \to \infty} \frac{Op(n)}{4^n} = \frac{1}{3}c$$

Der durchschnittliche Rechenaufwand pro Punkt konvergiert also zu einem konstanten Wert. Somit erhöht sich der absolute Aufwand bei Erhöhung des Grads um eins ungefähr um den Faktor vier, der durchschnittliche Aufwand pro Punkt aber gleich bleibt, da sich die Anzahl der Punkte mit Erhöhung des Grads ebenso vervierfacht.

3 Korrektheit

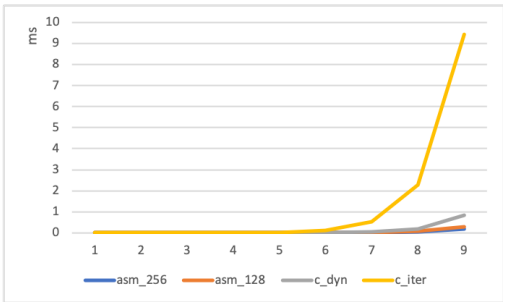
Im Gegensatz zu Aufgabenstellungen, die Floating-Point Berechnungen enthalten oder im Bereich der Kryptographie liegen, ist in unserem Fall eine Überprüfung der Korrektheit unserer Lösung zumindest in Graden < 10 allein schon durch das Betrachten der visuellen Repräsentation der Koordinaten im vom Rahmenprogramm erzeugten .svg File möglich. Automatisierte Tests können lediglich die erzeugten Koordinaten mit von einer anderen Implementierung berechneten Koordinaten abgleichen und feststellen, ob diese identisch sind, was nur bei einer bewiesen korrekten Vergleichsimplementierung aussagekräftig ist oder große Mengen an im Vorhinein gespeicherten Punktdaten erfordert. Der Nachweis der Korrektheit der Algorithmen ist also zunächst nur durch detailliertes Nachvollziehen ihrer Funktionsweise möglich, kann aber in niedrigeren Graden sehr gut visuell überprüft werden. Insbesondere da Aufrufe unserer Algorithmen für höhere Grade die Ergebnisse niedrigerer Grade, dessen Korrektheit noch gut

überprüfbar ist, dynamisch wiederverwenden, kann von einer insgesamten Korrektheit der Algorithmen ausgegegangen werden.

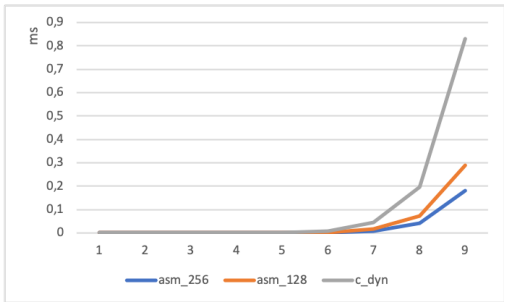
4 Performanzanalyse

Um das Laufzeitzeitverhalten unserer Implementierung der Moore-Kurve zu überprüfen und beurteilen, haben wir Benchmarks folgender vier Implementierungen auf einem MacBook Pro mit einem 3,1 Ghz Dual-Core Intel i7 und 16 Gb 1867 Mhz DDR3 RAM durchgeführt.

Grad n	# Punkte	Speicher	*asm_avx*	*asm*	*c_batch*	*c_iter*
3	64	512 B	27 ns	23 ns	67 ns	564 ns
6	4096	32 KiB	601 ns	1213 ns	6524 ns	106 μs
9	26144	2048 KiB	179 μs	289 μs	830 μs	9436 μs
12	16,78 $\cdot 10^6$	128 MiB	21,5 ms	24,5 ms	94,3 ms	705,6 ms
15	1,07 $\cdot 10^9$	8 GiB	1575,9 ms	1607,4 ms	7108,1 ms	50759 ms
16	4,29 $\cdot 10^9$	32 GiB	108,3 s	125,7 s	175,9 s	442,8 s



(a) alle Implementierungen



(b) ohne *c_iter*

Abbildung 8: Laufzeitverhalten der vier Implementierungen bis Grad 9

4.1 Iterative Berechnung der Moore-Koordinaten

Aus Abb. 8 geht klar hervor, dass die Punkt-für-Punkt-Berechnung der Moore-Koordinaten (wie in Sektion 2 beschrieben) deutlich ineffizienter ist als die anderen Implementierungen ist. Ein Vorteil dieser Variante ist zwar, dass sie nur sehr wenige Speicherzugriffe benötigt, da alle Berechnungen in den Registern durchgeführt werden können und nur das Endergebnis in den Speicher geschrieben wird. Durch das rein iterative Vorgehen kann dieser Algorithmus jedoch nicht sinnvoll auf Datenebene parallelisiert werden. Außerdem kann der dynamische Ansatz, von dem die übrigen Implementierungen profitieren, hier nicht angewandt werden. Im folgenden liegt daher der Fokus auf den restlichen Implementierungen.

4.2 Vergleich Assembler- mit Referenzprogramm

An Abb. 8.b und den Benchmarkwerten kann man erkennen, dass die Assembler Implementierungen deutlich performanter sind als die C-Referenzimplementierung. Für Grad 6 ist die Assembler-Implementierung, welche 128-bit Register benutzt, fünf mal schneller als das C-Programm. Dies ist darin begründet, dass sich die dynamische Berechnung der Hilbert-Kurve optimal durch SIMD parallelisieren lässt. Somit können einige Koordinaten gleichzeitig innerhalb einer Operation verarbeitet werden. Die Anzahl der Punkte, die in ein SIMD-Register passen ist abhängig vom Datentypen der Koodinaten und der Größe der Vektorregister. Wir haben uns entschieden, von der in der Aufgabenstellung vorgegebenen Signatur `moore(uint64_t degree, uint64_t *x, uint64_t *y)` abzuweichen und 32 bit Integer zu benutzen. Der maximal darstellbare Wert verringert sich dadurch zwar auf $2^{32} - 1$, jedoch werden größere Werte erst ab einem $n > 32$ benötigt. Der Speicherbedarf einer Moore-Kurve mit Grad 33 betrüge $4^{33} \cdot 2 \cdot 4$ Bytes $\approx 590 \cdot 10^{18}$ Bytes $= 590 \cdot 10^{6}$ Terrabytes, wenn man alle Koordinaten als 32 bit unsigned Integers darstellen könnte und das Doppelte bei korrekter Benutzung von 64-bit Integern. Aus diesem Grund ist es praktisch unmöglich, alle Punkte der Moore-Kurve 33. Grades zu speichern und die Nutzung von 32-bit Integern anstelle von `uint64_t` bringt realistisch gesehen keine Einschränkungen mit sich.
Durch die Reduzierung auf 32-bit Intger können nun bei Verwenung von 128 bit Vektorregistern vier und bei 258-bit Registern acht Werte gleichzeitig verarbeitet werden (siehe Abb. 9).

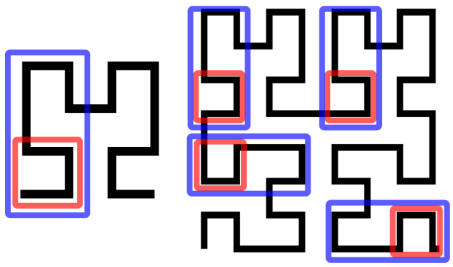
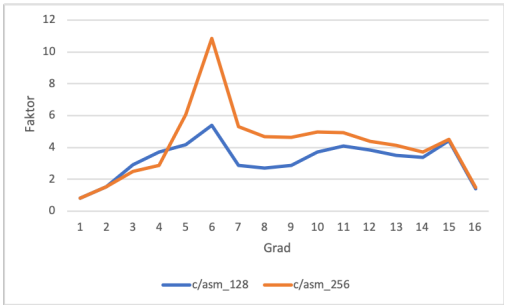


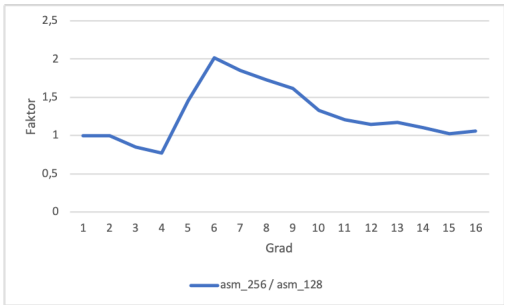
Abbildung 9: Parallelisierung durch SIMD

Durch Disassemblieren der mit `-march=native -O3` kompilierten Binärdatei haben wir festgestellt, dass der Compiler den C-Code nicht vektorisiert. Daher war unsere anfängliche Vermutung, dass *asm_128* (unsere Assembler-Implementierung, die 128-Bit Register Benutzt) etwa vier mal so schnell ist wie unsere C-Implementierung und *asm_256* (256-Bit AVX-Register) in der Hälfte der Zeit von *asm_128* terminiert.

Abb. 10.a bestätigt den ersten Teil unserer Vermutung. Zwischen Grad 4 und 15 ist die *asm_128* Implementierung 2,8 bis 5,3 mal schneller als die Referenzimplementierung in C. Der zweite Teil trifft nur teilweise zu. Wie in Abb. 10.b zu sehen ist, profitieren vor allem Grade zwischen 6 und 10 von den größeren Vektorregistern. Bei größeren Graden bietet die Verwendung von AVX-Registern keine signifikante Performanzsteigerung.



(a) Speedup Assembler- ggü. C-Implementierung

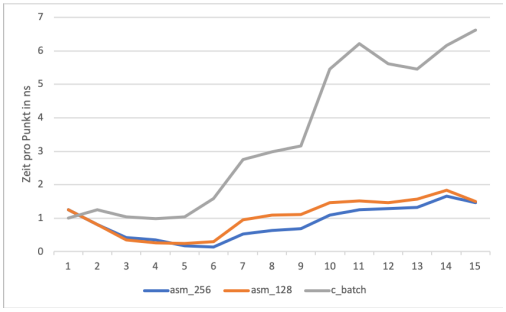


(b) Speedup *asm_256* ggü. *asm_128*

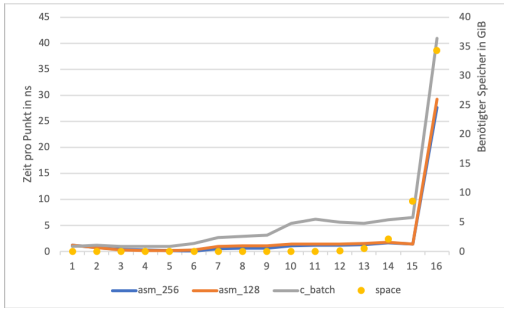
Abbildung 10: Laufzeitvergleich C/ASM

4.3 Cache Effizienz

Abb. 10.a wirft die Frage auf, weshalb die Assembler Implementierug bei Grad 6 besonders performant ist. Dies lässt sich vermutlich damit beantworten, dass die Moore-Kurve sechsten Grades genau *32 KiB* an Speicher benötigt und die Größe des L1 Caches der Maschine, auf der getestet wurde *32 KiB* beträgt. Um diese Annahme zu überprüfen wurde das Programm mit dem MacOS Tool *Instruments* auf Cache-Misses (insbesondere L1) überprüft, mit dem Ergebnis, dass 6 der höchste Grad ist, bei dem noch nahezu keine L1 Misses auftreten.



(a) Grad 1 - 15



(b) Grad 1 - 16

Abbildung 11: Nanosekunden pro Punkt der Moore-Kurve

Der Vollständigkeit halber muss noch der Effizienzeinbruch bei Grad 16 angesprochen werden. Dieser ist damit zu erklären, dass die Speicheranforderung für die Berechnung der Moore-Kurve auf 32GiB steigt. Da unser Computer nur 16 GiB an Arbeitsspeicher zur Verfügung stellt, müssen nun häufig Speicherseiten auf die Festplatte ausgelagert werden, was zu starken Performanzeinbußen führt.

5 Zusammenfassung und Ausblick

Selten gibt es einen Punkt, ab dem man mit Sicherheit die Möglichkeit weiterer Performanzoptimierungen eines Programms ausschließen kann. Besonders beim Schreiben von Assemblycode können so viele Faktoren wie eine simple Änderung der Instruktionsreihenfolge die Schnelligkeit des Programms beeinflussen, sodass die Annahme eines voll optimierten Programms nie getroffen werden kann.
Nach wachsender Einarbeitung in die Thematik sowie Entwicklung und ständiger Verbesserung unserer Algorithmen fallen uns zu diesem Zeitpunkt noch weitere mögliche Wege der Optimierung ein.
Unsere Benutzung von 32-Bit Integern statt 64-Bit-Integern zur Darstellung der Koordinaten hat zu einer höheren Parallelisierbarkeit des Programms geführt. Mit hoher Wahrscheinlichkeit kann für Grade ≤ 16 die dynamische Anpassung der Datenbreite auf 16 Bit zu einem weiteren Performanzgewinn führen. In Verbindung mit der Benutzung noch breiterer Register, zum Beispiel 512 Bit AVX Registern können Berechnungen noch stärker parallelisiert werden, wobei sich die Limitierung derartiger Optimierungen schon bei der Erhöhung von 128 Bit auf 256 Bit gezeigt hat, mit teilweise kaum besserer Perfomanz trotz breiterer Register.
Auch das Experimentieren mit einer veränderten Datenstruktur, die x- und y-Koordinate eines Punktes nah beieinander speichert, könnte in bestimmten Fällen die Performance durch eine verbesserte Reihenfolge von Lade- und Schreibbefehlen auf den Speicher erhöhen.
All dies kann in unserer Herangehensweise jedoch nur den konstanten Aufwand pro Punkt verringern. Einen Algorithmus zu entwickeln, der sich für den gesamten Aufwand nicht in einer exponentiellen Laufzeitklasse befindet ist allein schon durch die exponentiell ansteigende Punktanzahl und somit einer exponentiell steigenden Anzahl an Schreibbefehlen unmöglich.

6 Anhang: Algorithmen

Algorithm 1 Berechne Koordinaten eines Punktes auf Moore-Kurve

function BERECHNEMOOREKOORDINATE($index$, $degree$)
  finde Quadrant q des Punkts q heraus ($q \in \{A, B, C, D\}$)
  $coord \leftarrow$ berechneHilbertKoordinate($index$, $degree - 1$)
  if $q = A$ **then** rotiere $coord$ um 90° ↺
  else if $q = B$ **then** rotiere $coord$ um 90° ↺ und verschiebe nach oben
  else if $q = C$ **then** rotiere $coord$um 90° ↻ und verschiebe nach oben und nach rechts
  else if $q = D$ **then** rotiere $coord$ um 90° ↻ und verschiebe nach rechts
  end if
end function
function BERECHNEHILBERTKOORDINATE($index$, $degree$)
  [...]
end function

Algorithm 2 Berechne Koordinaten eines gegebenen Punkts auf der Hilbert-Kurve

function BERECHNEHILBERTKOORDINATE($index$, $degree$)
  $sidelength \leftarrow 2^{degree}$
  $i \leftarrow 1$
  $coord \leftarrow (0, 0)$
  while $i < sidelength$ **do**
    ermittle Quadrant $q \in \{A, B, C, D\}$ mit den letzten zwei Bits von $index$:
    $A \iff$ '00', $B \iff$ '01', $C \iff$ '10', $D \iff$ '11'
    if $q \in \{A, D\}$ **then**
      if $q = D$ **then** ▷ falls sich Punkt in einem der beiden oberen Quadranten befindet
        spiegle $coord$ an $y = i - 1 - x$
      else
        spiegle $coord$ an $y = x$ Achse/Winkelhalbierender
      end if
    else
      verschiebe $coord$ nach oben, denn $q \in \{B, C\}$
    end if
    if $q \in \{C, D\}$ **then**
      verschiebe $coord$ nach rechts
    end if
    $index \leftarrow \frac{index}{4}$ ▷ zweimaliger rechts-Bitshift um nächste 2 Bits anzusehen
    $i \leftarrow 2i$ ▷ Die Seitenlänge verdoppelt sich mit jeder Iteration
  end while
  return $coord$
end function

Algorithm 3 Berechne alle Punkte der Moore-Kurve

function BERECHNEMOOREPUNKTE($degree$, int[] x, int[] y)
  Koordinaten für $degree = 1$ in x- und y-Arrays hardcoden (jeweils 4 Werte)
  if $degree = 1$ **then return**
  end if
  berechneHilbertPunkte($degree - 1$, int[] x, int[] y) ▷ schreibt Pkt. der H. Kurve Grad $degree - 1$ in die Arrays
  $len \leftarrow 2^{degree-1} - 1$ ▷ Offset für Translation
  $quarter \leftarrow 4^{degree-1}$ ▷ Seitenlänge der Hilbert-Kurve quadriert $= (2^{degree-1})^2$
  for $i \leftarrow 0$ to $quarter - 1$ **do**
    Quadrant B: rotiere i-ten Punkt in A um 90° ↻ und verschiebe um len nach oben
    Quadrant C: rotiere i-ten Punkt in A um 90° ↻ und verschiebe um len nach oben und rechts
    Quadrant D: kopiere i-ten Punkt in C und verschiebe um len nach unten
    Quadrant A: kopiere i-ten Punkt in B and verschiebe um len nach unten
  end for
end function

function BERECHNEHILBERTPUNKTE($degree$, int[] x, int[] y)
  $len \leftarrow 2$ ▷ Offset beginnt bei 2
  $quarter \leftarrow 4$ ▷ Algorithmus beginnt mit 4 hardgecodeten Punkten
  for $j \leftarrow 2$ to $degree$ **do** ▷ Die Erzeugung beginnt bei Grad 2, da Grad 1 bereits im Speicher steht
    for $i \leftarrow 0$ to $quarter - 1$ **do**
      Quadrant B: kopiere und verschiebe i-ten Punkt in A um len nach oben
      Quadrant A: spiegle i-ten Punkt in A an Winkelhalbierender $x = y$
      Quadrant C: kopiere i-ten Punkt in B und verschiebe um len nach oben
      Quadrant D: rotiere i-ten Punkt in A um 180° und verschiebe um len nach rechts
    end for
    $len \leftarrow 2 * len$ ▷ mit jeder Iteration verdoppelt sich die Seitenlänge..
    $quarter \leftarrow 4 * quarter$ ▷ ...und vervierfacht sich die Anzahl der Punkte
  end for
end function

Literatur

[1] Nick Berry. Hilbert curves, March 2013.

[2] John Burkardt. Convert between 1d and 2d coordinates of hilbert curve, December 2015.

[3] Marcin Chwedczuk. Iterative algorithm for drawing hilbert curve, August 2016.

[4] Robert Dickau. Stages 0 through 5 of fractal moore curve, May 2008.

[5] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, April 2016. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, visited 2017-08-19.

[6] Sven Kreiss. S2 cells and space-filling curves: Keys to building better digital map tools for cities, July 2016.

[7] Eliakim Hastings Moore. On certain crinkly curves. *Transactions of the American Mathematical Society*, 1(1):72–90, 1900.

[8] Christian S. Perone. Google's s2, geometry on the sphere, cells and hilbert curve, August 2015.