Mika Auer

Hochschule Karlsruhe | Informatik (Master)

Optimierung von Programmen

Bericht

Im Folgenden finden Sie das Berichtsheft zu den Aufgaben der Vorlesung „Optimierung von Programmen“ bei Prof. Dr. Christian Pape. Die Aufgaben wurden von dem Masterstudenten Mika Auer bearbeitet. Dabei wurde ein Programm bereitgestellt, dass mittels Raytracing ein 2-dimensionales Bild einer Teekanne erstellt. Dieses Programm soll nun von den Studenten optimiert werden. In den folgenden 3 Aufgaben finden sich dazu Optimierungen, die an verschiedenen Stellen im Code ansetzen, um die Ausführung des Programms zu beschleunigen. Für die Kompilierung wurde der G++ Compiler in Version 7.5.0 verwendet. Da dieser nicht nativ auf der zur Entwicklung verwendeten Maschine unterstützt wird, wird dieser über das offizielle GCC-Docker-Image ausgeführt.

# Aufgabe 1

In der ersten Aufgabe gab es mehrere vorgegebene Optimierungen die zu implementieren waren. Zuerst sollte eine Abbruchbedingung für die Schnittpunktbestimmung eingebaut werden. Beim Raytracing wird die zu verwendende Farbe und Helligkeit eines Pixels dadurch bestimmt, dass man Strahlen aus einem Sichtpunkt und einer Lichtquelle verfolgt und deren Schnittpunkte mit den Objekten der Szene bestimmt. In der nicht optimierten Variante werden diese Schnittpunkte für jeden einem Objekt zugehörigen Punkt der Szene bestimmt. In der optimierten Variante macht man sich nur zunutze, das Strahlen, die aus demselben Quellpunkt in dieselbe Richtung gehen, nur das erste auf diesem Weg getroffene Objekt treffen und der zugehörige Pixel diese Farbe annehmen sollte. Punkte der Szene die von einem Stahl aus derselben Quelle (im Sichtpunkt) in dieselbe Richtung zeigen, sind nur dann sichtbar, wenn sich auf diesem Stahl kein Objekt näher zum Sichtpunkt befinden. Dies macht man sich nun zunutze, indem man bei der Schnittpunktberechnung mit der Formel **p = ray.origin + t \* ray.direction** den minimalen Wert von t speichert und die Berechnung abbricht, wenn sie mit einem Wert größer  **minimum\_t** durchgeführt wird. Dadurch kann man sich einige aufwändige Schnittpunktberechnungen sparen, die später nicht in das Endergebnis einfließen. Diese Optimierung wurde in der Methode „**intersects**“ der Klasse „**Triangle**“ implementiert. Dort wurde nach der Berechnung des neuen Wertes von **t** eine Abbruchbedingung **t > minimum\_t** eingebaut, bei deren zutreffen die Methode mit der Rückgabe eines false-Bool Wertes beendet wird. Dieser Wert symbolisiert, dass kein neuer nähester Schnittpunkt gefunden wurde.

Text

Description automatically generated

Text

Description automatically generated

Insgesamt konnte in den Zeitmessungen durch diese Optimierung eine Minimierung der Ausführungszeit um ca. 3% festgestellt werden.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time in s | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Schnitt |
| Nicht optimiert | 8,674 | 8,720 | 8,648 | 8,666 | 8,743 | 8,341 | 8,637 | 8,537 | 8,531 | 8,343 | 8,548 |
| Mit Abbruchbedingung | 8,279 | 8,396 | 8,226 | 8,262 | 8,347 | 8,482 | 8,363 | 8,202 | 8,606 | 8,168 | 8,331 |

Wenn man das Programm mit gcov analysiert, fällt auch auf, dass durch den frühzeitigen Abbruch viele Durchläufe gespart werden. In der nicht optimierten Variante, läuft das Programm 458857871-mal durch die Schnittpunktbestimmung, nach der Berechnung von t, durch. In der Optimierten Variante nur noch 413970453-mal. Man kann also ungefähr 10% der Durchläufe frühzeitig abbrechen.

Eine weitere Optimierung, die vorgenommen werden sollte, war es die Berechnung der u-v-Parameter nach hinten zu verschieben und die Anzahl der vorgenommenen Quadratwurzelberechnungen von drei auf zwei zu reduzieren. In der dort aufgerufenen Methode **length()** wurde diese verwendet. Dabei hat man die Länge eines Vektors und der Normale berechnet und dann durch einander dividiert. Man konnte die Anzahl der Wurzelberechnungen nun reduzieren, indem man nicht **length()** sondern **squareOfLength()** aufruft und dann erst nach der Division die Wurzel zieht. So wird eine der Rechenaufwändigen Wurzelberechnungen eingespart. Durch das Verschieben der Berechnung von **area** bzw. **square\_area** weiter nach hinten im Code, kurz bevor der Wert gebraucht wird, kann man die Anzahl der Berechnungen dieses Wertes von 519950720 auf 104877228 reduzieren. Man hat also nur noch ca. 20% dieser Aufrufe, bei denen auch eine Quadratwurzel berechnet wird.

A screenshot of a computer

Description automatically generated with medium confidenceText

Description automatically generated

Nach Messung der Zeiten in dieser optimierten Variante konnten das Programm durch das Weglassen der einen Wurzelberechnung um ganze 7% beschleunigt werden. Dafür das es sich um eine einzige Berechnung handelt, die hier eingespart wird, ist dies ein sehr beachtliches Ergebnis. Die Berechnung einer Quadratwurzel ist offenbar sehr aufwändig. Insgesamt konnte das Programm durch die Gesamtheit der hier genannten Optimierungen um ca. 11 % beschleunigt werden.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time in s | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Schnitt |
| Ohne Optimierungen | 8,674 | 8,720 | 8,648 | 8,666 | 8,743 | 8,341 | 8,637 | 8,537 | 8,531 | 8,343 | 8,584 |
| Optimierung Wurzelberechnung | 7,938 | 7,986 | 8,008 | 8,069 | 7,913 | 8,075 | 7,973 | 7,964 | 8,017 | 7,822 | 7,9765 |
| Alle Optimierungen | 7,525 | 7,559 | 7,661 | 7,597 | 7,670 | 7,691 | 7,787 | 7,624 | 7,537 | 7,578 | 7,6229 |

Eine Optimierung die der Compiler selbst vorgenommen hat, ist die Verwendung von SIMD Befehlen. Im Screenshot kann man sehen, dass z.B. die Befehle **vfmadd132ss** und **vdivss** genutzt werden. Dadurch entsteht eine zusätzliche Parallelisierung auf Prozessorebene, bei der mehrere Berechnungen gleichzeitig durchgeführt werden. In diesem Fall wird z.B. die Berechnung der **square\_of\_length** parallelisiert. Die SIMD Befehle werde eingebaut, da wir beim kompilieren des Programms mit den Optionen **-mfpmath=sse -mavx** explizit angegeben haben, dass diese verwendet werden sollen.

Graphical user interface, text

Description automatically generated

Eine weitere denkbare Optimierung wäre, die Berechnung von **square\_u** und **square\_area** ganz nach hinten zu **square\_v** zu verschieben, indem man den zu u zugehörigen Vektor zwischenspeichert. Die Idee dahinter ist, dass der Compiler die dann untereinanderstehenden Berechnungen zu gemeinsamen SIMD-Befehlen Optimieren könnte. Leider ist dies aber nur eine Idee und nach austesten dieser Codeveränderung kommt es weder zu wesentlichen Veränderungen im Assembler-Code noch zu einer beobachtbaren Beschleunigung des Programms. (Dieses war sogar minimal langsamer, was aber an minimalen Veränderungen der Umgebung liegen kann)

Assembler Code der optimierten Codestellen:

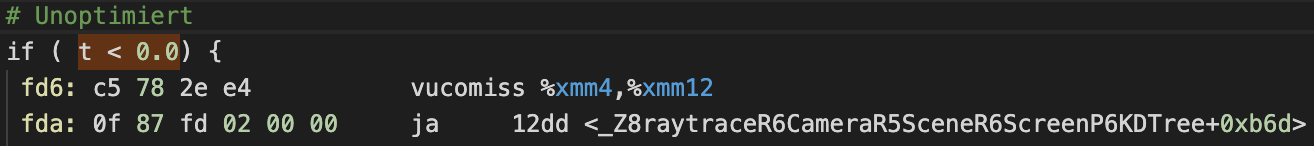
Optimiert:

Abbruch Minimum_t


Text

Description automatically generated

Unoptimiert:



Text

Description automatically generated