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

Text

Description automatically generatedIn 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

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:

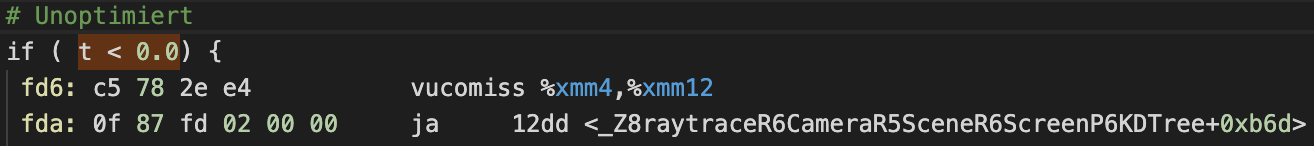
Optimiert:

Abbruch Minimum_t


Text

Description automatically generated

Unoptimiert:



Text

Description automatically generated

# Aufgabe 2

Das zweite Aufgabenblatt betrachtet die Approximierung der Quadratwurzelberechnung mit Hilfe des Newton-Verfahrens. Das Newton-Verfahren verwendet eine über mehrere Iterationen nutzbare mathematische Gleichung, die sehr nahe an die Berechnung der realen Quadratwurzel herankommt. Je mehr Iterationen desto genauer, aber auch desto länger die Rechenzeit. Braucht man nur eine einfache Näherung des Wertes der Quadratwurzel eines Input-Parameters, kann man das Verfahren mit wenigen Iterationen verwenden und damit die Berechnung deutlich beschleunigen.

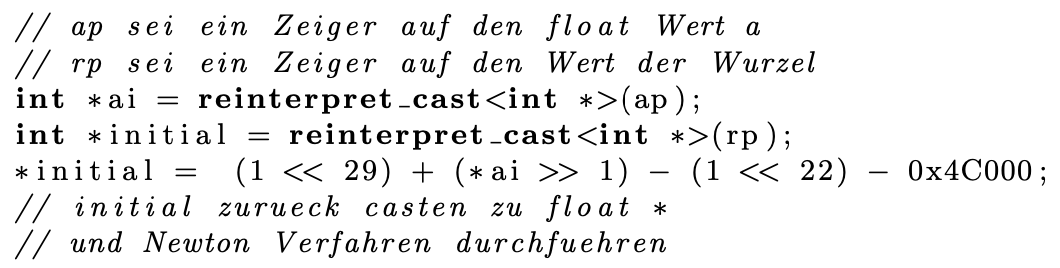


Abbildung Berechnung des Initialwertes von xn

A picture containing text

Description automatically generated

Abbildung 2Funktion des Newton-Verfahrens

In der Aufgabe war gefordert mehrere Implementierungen dieses Verfahrens auszuprobieren und in ihrer Laufzeit zu vergleichen. Dazu war jeweils zuerst nötig die Input Parameter zu einer Ganzzahl zu wandeln und einige Shiftoperationen, eine Addition und eine Division auf diese anzuwenden, damit ein initialer Startwert für xn bestimmt werden konnte. Von diesem Stand aus müssen die Input-Werte in ihre ursprüngliche Form zurückgewandelt werden und die unterschiedlichen Implementierungen gehen dann mit diesem Startwert um.

In der ersten Teilaufgabe war gefordert die Berechnung für einen einfachen float-Wert durchzuführen und die Funktion so zu implementieren, wie sie für eine einzige Zahl anzuwenden ist. Die zweite Teilaufgabe sollte dann damit experimentieren, wie es sich auswirkt, wenn man mehrere Werte in aufeinanderfolgenden Zeigern als Input übergibt. Man teilt dies dem Compiler über das „\_\_restrict\_\_“ Keyword mit. Es soll nun beobachtet werden inwiefern der Compiler dies nutzt, um das Programm zu beschleunigen. Die dritte Teilaufgabe beschäftigt sich dann damit, dass dem Compiler explizit ein Vector-Datentyp (v4sf) übergeben wird. Dieser wird in der Datei sqrt\_opt.h definiert und entspricht einem Vector von Floats mit einer Gesamtlänge von 16 Bytes, also 4 Float-Werten. Dadurch teilt man dem Compiler explizit mit, dass er diese 4 Werte zusammen in einem Schritt berechnen kann. Auch hier soll beobachtet werden, was für einen Einfluss dies auf die Laufzeit und den Assembler-Code hat. In beiden Fällen wird erwartet, dass der Compiler SIMD-Befehle verwendet.

## Sqrt1

Die einfache Verwendung des Newton Verfahrens bringt bei einer geringen Anzahl an Iterationen eine deutliche Beschleunigung des Programms. Je nachdem ob man mit einer inneren Schleife, die jede Wurzel einzeln berechnet oder einer Schleife die 4 in einer Iteration zu berechnen versucht, arbeitet, kann man bei 2 Iterationen eine 7,4-fache bzw. 6,3-fache Beschleunigung, bei 3 Iterationen eine 5,7-fache bzw. 5,4-fache und bei 4 Iterationen eine 4,4-fache bzw. 4,2-fache Beschleunigung gegenüber der Standard-Wurzelfunktion erzielen.

Text

Description automatically generated

Abbildung 3Implementierung sqrt1

Die Implementierung ist simpel. Der Input-Pointer a und der Pointer auf den Initialwert root werden zuerst zu Integer-Pointern gecastet, damit ihre Werte als Ganzzahlen behandelt werden. Dies ist nötig, um den Initialwert neu zu berechnen, mit dem man das Newton-Verfahren startet. Mit dem Verfahren aus Abbildung 1 wird der Initialwert von xn berechnet und in ri gespeichert. Dieser Wert wird anschließend wieder als float interpretiert und anschließend das Newton-Verfahren aus Abbildung 2 verwendet. Für die Anzahl der Iterationen wird eine Schleife verwendet, die die Anzahl aus einem Template-Parameter nimmt. Die Funktion aus dem Newton-Verfahren ist innerhalb dieser Schleife zu finden, wo sie root neu berechnet. Zum Schluss wird dieses root dann auch als Ergebnis zurückgeliefert.

Die Frage ist nun, warum der Aufruf aus Abbildung 4 eine schnellere Wurzelberechnung umsetzt als der Aufruf aus Abbildung 5? Dazu muss der erzeugte Assembler-Code betrachtet werden. In Abbildung 7 ist der Aufruf aus der Schleife über alle Elemente zu sehen und in Abbildung 8 der Aufruf aus der 4er-Schleife. Auffällig ist dabei, dass bei letzterem deutlich mehr Operationen nötig sind als bei ersterem. Es wird anscheinend einiges für den Aufruf vorbereitet, bevor es schlussendlich zu diesem kommt. Die Funktion ist bei beiden Aufrufen gleich (wie in Abbildung 6sqrt1 Assembler zu sehen), jedoch dauert die Berechnung durch die kompliziertere Verarbeitung vor dem Aufruf in der 4er-Schleife dort etwas länger. Je mehr Iterationen man allerdings beim Newton-Verfahren verwendet, umso geringer wird dieser Vorteil relativ zur Gesamtlaufzeit. Auffällig im Assembler Code ist, dass alle Berechnungen schon für SIMD-Befehle optimiert wurden (zu erkennen an den vielen Vektor-Befehlen mit „v“ am Anfang).

Text

Description automatically generated

Abbildung Aufruf von sqrt in Schleife über alle Objekte

A screenshot of a computer

Description automatically generated with medium confidence

Abbildung Aufruf von sqrt1 in innerer 4er-Schleife

Text

Description automatically generated

Abbildung 6sqrt1 Assembler

Text

Description automatically generated

Abbildung 7sqrt1 Aufruf einzeln Assembler

Text

Description automatically generated

Abbildung 8sqrt1 Aufruf 4er-Schleife Assembler

## Sqrt2

Calendar

Description automatically generated with medium confidenceDie Implementierung aus Teilaufgabe 2 ähnelt der aus Teilaufgabe 1. Der große Unterschied ist nun das wir als Übergabeparameter an die Funktion einen Zeiger übergeben, der den Beginn einer Reihe von 4 aufeinanderfolgenden float Werten markiert. Zudem Übergeben wir einen Zeiger auf die Position, an der die Ergebnisse der 4 Berechnungen gespeichert werden sollen. Die Berechnungen der 4 Werte werden nun im Code untereinander angeordnet. Man versucht so dem Compiler mitzuteilen, dass diese zusammengehören und auch zusammen berechnet werden können. Man erhofft sich also, dass diese als SIMD-Befehle ausgeführt werden (werden sie aber offenbar in Teilaufgabe 1 sowieso schon). Durch das Keyword „\_\_restrict\_\_“ bei der Deklaration der Übergabeparameter, sagt man dem Compiler zudem, dass hier mehrere Werte übergeben werden, die Zeiger also nicht auf dieselben Speicherbereiche zeigen. Dies kann dem Compiler bei der Optimierung helfen.

Abbildung Implementierung sqrt2

Die erhoffte Beschleunigung des Programms bleibt aus. Zwar ist diese Variante minimal schneller als sqrt1 aufgerufen in der 4er-Schleife, aber langsamer als sqrt1 in der großen Schleife. Bei 2 Iterationen ist eine 6,8-fache, bei 3 Iterationen eine 5,4-fache und bei 4 Iterationen eine 4,2-fache Beschleunigung zu beobachten.

Graphical user interface, text

Description automatically generatedBei einer Betrachtung der Assembler Instruktionen fällt auf, dass der reinterpret\_cast deutlich komplizierter geworden ist. Dem Assembler code des ersten Teilstückes der Implementierung ist zu entnehmen, dass die Bitshifts im Assembler Code sehr warscheinlich den letzten beiden Befehlen dieses Stückes (vpsradd und vpaddd) entsprechen. Dies macht Sinn, da sich die Zeile mit dem Bitshift vereinfachen lässt zu einer Rechnung bei der man nur noch den Bitshift von ai und eine Addition mit dem Rest (wenn man diese festen Werte zusammenrechnet, bekommt man ein positives Ergebnis (532365312)). Somit entspricht der reinterpret\_cast in Teilaufgabe 1 nur einem einfach Verschiebungsbefehl (vmovdqa)

Graphical user interface

Description automatically generated with low confidenceUnd in Teilaufgabe 2 einer langen Folge von Verschiebung-, Subtraktions-, Shuffle und Permutationsbefehlen. Diese zusätzlichen Instruktionen wirken sich hier negativ auf die erhoffte Beschleunigung aus. Ein Vorteil ist aber, dass diese Instruktionen nur einmal pro 4er-Tupel durchgeführt werden müssen und danach für die restlichen 3 Werte nur noch die Bitshift und Addition durchgeführt werden muss. Trotzdem scheint dieser Vorteil die zuvor längere Folge von Instruktionen nicht auszugleichen.

Abbildung sqrt2 Cast und Initialwertberechnung Assembler

In Abbildung 11 sieht man dann die Assembler Instruktionen zur Berechnung des eigentlichen Ergebnisses des Newton Verfahrens und noch einmal die Schritte, die für die Berechnung des Initialwertes für die Berechnungen 2 bis 4. Hier sind klar die Additions- Multiplikations- und Divisionsbefehle zu erkennen. Es wurden wie erwartet SIMD-/AVX-Befehle verwendet. Es werden allerdings deutlich mehr Instruktionen verwendet und das Programm wird komplizierter. Dadurch kommt es auch dazu das es gegenüber der einfach sqrt1 zu keiner Beschleunigung kommt.

## Sqrt3

Teilaufgabe 3 forderte das Newton-Verfahren nun mit Hilfe von Vector-Datentypen, also Datentypen die speziell für die parallele Berechnung gemacht sind, umzusetzen. Dazu wurde definiert, dass v4sf einem Float-Vector mit einer Größe von 16Bytes/128-Bit entspricht und v4si einem Integer-Vector derselben Länge. Anstatt nun mit float und int zu arbeiten werden diese neu definierten Datentypen genutzt. Dadurch wird der Code wieder so kurz wie in Teilaufgabe 1 und verändert sich gegenüber dieser kaum, bis darauf das nun die neuen Datentypen genutzt werden und bei den Übergabeparametern wie in Teilaufgabe 2 auch eine Rückschreibeadresse mitgegeben wird, anstatt auf die Nutzung von „return“ zu setzen.

Abbildung 11Newton Verfahren sqrt2 Assembler Code

Text

Description automatically generated

Abbildung sqrt3 Implementierung

Bei den Ausführungszeiten macht sich die Optimierung nicht wirklich bemerkbar. Zwar ist bei zwei Iterationen eine durchschnittliche Beschleunigung gegenüber der System-Wurzelberechnungsfunktion vom Faktor 7 zu beobachten, was auch stärker ist als bei sqrt2 und sqrt1 innerhalb der 4er-Schleife, bei mehr Iterationen verschwindet dieser Vorteil aber gänzlich. Bei 3 Iterationen ist nur noch eine 4,6-fache und bei 4 Iterationen eine 3,2-fache Beschleunigung gegenüber der Standardfunktion zu beobachten. Es wurden noch weitere Messungen in einem Poolraum der Hochschule (LI137) auf einem Rechner mit Ubuntu und G++ v.9 und nativ auf demselben Computer auf MacOS 12.0.1 mit clang 11.0.0 durchgeführt, bei denen diese Variante deutlich bessere Performance zeigt als die anderen Implementierungen. Da aber zu Beginn dieses Labors gesagt wurde das G++ v.7 oder älter verwendet werden soll, wurden diese Ergebnisse hier nicht mit in die Evaluation aufgenommen.

Text

Description automatically generatedDer Assembler Code von sqrt3 ähnelt dem von sqrt1 sehr (bzw. ist nahezu identisch). Dies liegt daran, dass der Compiler schon bei sqrt1 automatisch SIMD-Befehle verwendet hat. Beim reinterpret\_cast werden die Werte einfach an ihre neue Adresse verschoben (vmovdqa) und dann bei der Berechnung des Initialwertes wird ai um 1 Bit nach rechts rotiert und im Folgenden mit dem Ergebnis der restlichen Werte addiert (vpsrad, vpaddd). Bei der Berechnung der einzelnen Iterationen des Newton-Verfahres werden wie zu erwarten wieder Divisionen, Additionen und Multiplikationen (vdivps, vaddps, vmulps) durchgeführt. Im Assembler ist auch zu erkennen warum diese Variante langsame ist als die ersten Beiden. Die ersten Beiden wurden vom Compiler so optimiert, dass die ymm-Register genutzt werden. Dies sind 256-Bit-Register die für Vektorberechnungen genutzt werden können. In 256-Bit passen 8 Floats. Es können also auch 8 Berechnungen gleichzeitig durchgeführt werden. In der Variante in sqrt3 wurden die Vektortypen explizit angegeben und mit 16Byte, also 128-Bit definiert. Dadurch werden hier auch die 128-Bit xmm-Register verwendet. In diesen können dann nur 4 Float Berechnungen gleichzeitig durchgeführt werden, wodurch das Programm auch langsamer wird.

Abbildung sqrt3 Assembler

## Zusammenfassung der Ergebnisse

Insgesamt lässt sich Zusammenfassen, dass der Compiler von sich aus schon viel zu optimieren scheint. Die Ergebnisse sind je nach Compiler Version und System sehr unterschiedlich und lassen sich schwer miteinander vergleichen. Die beschriebenen Annahmen basieren auf den Ergebnissen und dem Assembler Code der bei der Nutzung in einer GCC:7 Dev-Container Umgebung mit G++ v7.5.0 beobachtet und erzeugt wurden.

Aus diesen Ergebnissen lässt sich Zusammenfassen, dass die Verwendung des Newton-Verfahrens die Quadratwurzelberechnung um ein Vielfaches beschleunigen kann. Weitere Optimierungen die die Parallelisierung des Programms fördern sollen, werden größtenteils vom Compiler selbst vorgenommen und können bei expliziter Deklaration sogar das Programm verlangsamen. Insgesamt ist das Newton-Verfahren zu empfehlen, weitere Optimierungen sind von System zu System auf ihren Nutzen zu überprüfen.

Messungen auf MacBook Pro 13-inch 2017 3,1GHz Dual-Core Intel Core i5 16GB DDR3 Ram in DevContainer Umgebung auf Docker-Image Gcc:7 mit G++ Version 7.5.0:

2 Iterationen:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time in ns | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | **Schnitt** | **Beschleunigung gegenüber Systemfunktion** |
| math sqrt | 2172803 | 1863940 | 1497120 | 1463831 | 1485728 | 2039605 | 1468095 | 1462728 | 1480265 | 1478777 | 1641289,2 |  |
| sqrt1 (1 time per loop) | 482104 | 218247 | 191873 | 175967 | 174590 | 266853 | 172809 | 178125 | 189975 | 173969 | 222451,2 | 7,378198904 |
| sqrt1 (4 times per loop) | 522775 | 262814 | 235796 | 216405 | 214278 | 246189 | 220207 | 213846 | 249906 | 225218 | 260743,4 | 6,294652904 |
| sqrt2 | 378790 | 270220 | 236233 | 211695 | 213465 | 216155 | 213567 | 211275 | 238918 | 227323 | 241764,1 | 6,788804459 |
| sqrt3 | 358771 | 239666 | 228483 | 210448 | 206470 | 214856 | 211165 | 211348 | 224896 | 222264 | 232836,7 | 7,049100077 |

3 Iterationen:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time in ns | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | **Schnitt** | **Beschleunigung gegenüber Systemfunktion** |
| math sqrt | 2188351 | 1480443 | 1478165 | 1473242 | 1461594 | 1486840 | 1460457 | 1506998 | 1541161 | 1478373 | 1555562,4 |  |
| sqrt1 (1 time per loop) | 367471 | 271766 | 268156 | 249186 | 252413 | 257650 | 256885 | 253746 | 264485 | 264669 | 270642,7 | 5,747660661 |
| sqrt1 (4 times per loop) | 417059 | 299725 | 275200 | 278409 | 260821 | 271280 | 268708 | 263328 | 286612 | 276402 | 289754,4 | 5,368554886 |
| sqrt2 | 403225 | 286354 | 285688 | 269176 | 266277 | 271912 | 265474 | 279019 | 286331 | 270581 | 288403,7 | 5,393697792 |
| sqrt3 | 472474 | 339029 | 331250 | 316157 | 309614 | 319676 | 317361 | 313236 | 315944 | 322187 | 335692,8 | 4,633886696 |

4 Iterationen:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time in ns | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | **Schnitt** | **Beschleunigung gegenüber Systemfunktion** |
| math sqrt | 2086571 | 1459637 | 1464037 | 1458379 | 1456588 | 1477957 | 1470399 | 1461446 | 1465910 | 1502986 | 1530391 |  |
| sqrt1 (1 time per loop) | 512475 | 334105 | 342764 | 332268 | 329350 | 330718 | 335593 | 330544 | 328189 | 340908 | 351691,4 | 4,351516699 |
| sqrt1 (4 times per loop) | 553702 | 351302 | 354015 | 336079 | 345430 | 343048 | 345332 | 340496 | 345498 | 352262 | 366716,4 | 4,173227595 |
| sqrt2 | 548376 | 357015 | 356066 | 340115 | 336647 | 334696 | 346015 | 339231 | 335866 | 351119 | 364514,6 | 4,198435399 |
| sqrt3 | 737127 | 454496 | 453851 | 440091 | 442101 | 442128 | 452765 | 438331 | 444720 | 463563 | 476917,3 | 3,208923224 |

Messungen auf PC in Poolraum LI137 mit Ubuntu und G++ v.9:

2 Iterationen:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time in ns | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | **Schnitt** | **Beschleunigung gegenüber Systemfunktion** |
| math sqrt | 808272 | 726898 | 729141 | 726844 | 742231 | 727156 | 727785 | 727911 | 726862 | 727206 | 737030,6 |  |
| sqrt1 (1 time per loop) | 1477 | 1462 | 1503 | 1473 | 1536 | 1442 | 1548 | 1462 | 1442 | 1464 | 1480,9 | 497,6909987 |
| sqrt1 (4 times per loop) | 389434 | 357788 | 357210 | 357828 | 365870 | 358469 | 357137 | 356690 | 356971 | 357439 | 361483,6 | 2,03890467 |
| sqrt2 | 119936 | 103473 | 103319 | 103492 | 105487 | 103711 | 103601 | 103102 | 103250 | 103786 | 105315,7 | 6,9982975 |
| sqrt3 | 429 | 396 | 376 | 395 | 408 | 395 | 399 | 421 | 375 | 377 | 397,1 | 1856,032737 |

3 Iterationen:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time in ns | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | **Schnitt** | **Beschleunigung gegenüber Systemfunktion** |
| math sqrt | 804093 | 727223 | 726717 | 730070 | 728645 | 727517 | 726170 | 725997 | 727375 | 725655 | 734946,2 |  |
| sqrt1 (1 time per loop) | 2361 | 2200 | 2188 | 2161 | 2208 | 2212 | 2170 | 2167 | 2209 | 2191 | 2206,7 | 333,0521593 |
| sqrt1 (4 times per loop) | 614938 | 544296 | 543317 | 543423 | 548048 | 543977 | 543187 | 544701 | 545536 | 543811 | 551523,4 | 1,332574828 |
| sqrt2 | 614587 | 535605 | 535683 | 536367 | 536705 | 539666 | 536632 | 535744 | 537604 | 536428 | 544502,1 | 1,349758247 |
| sqrt3 | 604 | 571 | 573 | 572 | 571 | 572 | 573 | 572 | 572 | 572 | 575,2 | 1277,722879 |

4 Iterationen:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Time in ns | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | **Schnitt** | **Beschleunigung gegenüber Systemfunktion** |
| math sqrt | 798157 | 729236 | 726908 | 728629 | 726925 | 727276 | 727785 | 727998 | 727585 | 726584 | 734708,3 |  |
| sqrt1 (1 time per loop) | 3175 | 2874 | 2913 | 2874 | 2986 | 2892 | 2872 | 2873 | 2871 | 2892 | 2922,2 | 251,4230032 |
| sqrt1 (4 times per loop) | 810978 | 727848 | 732032 | 728961 | 729800 | 730410 | 729243 | 728587 | 728101 | 729491 | 737545,1 | 0,996153727 |
| sqrt2 | 784596 | 714881 | 715275 | 716417 | 714238 | 713579 | 716246 | 714751 | 715098 | 716948 | 722202,9 | 1,017315632 |
| sqrt3 | 850 | 751 | 752 | 752 | 733 | 753 | 752 | 751 | 772 | 753 | 761,9 | 964,3106707 |