

Bericht Optimierung

Lukas Panni

Bericht Optimierung

1. Schnittpunkttest optimieren

Sourcecode optimierte Version:

Listing 1: intersects optimiert

```
bool intersects(Vector<T, 3> origin, Vector<T, 3> direction,
FLOAT &t, FLOAT &u, FLOAT &v, FLOAT minimum_t)
{
    Vector<T, 3> normal = cross_product(p2 - p1, p3 - p1);

    T normalRayProduct = normal.scalar_product(direction);

    if (fabs(normalRayProduct) < EPSILON)
    {
        return false;
    }

    T d = normal.scalar_product(p1);
    t = (d - normal.scalar_product(origin)) / normalRayProduct;

    if (t < 0.0 || t > minimum_t)
    {
        return false;
    }

    Vector<T, 3> intersection = origin + t \* direction;

    Vector<T, 3> vector = cross_product(p2 - p1, intersection - p1);
    if (normal.scalar_product(vector) < 0.0)
    {
        return false;
    }

    vector = cross_product(p3 - p2, intersection - p2);
```

```

if (normal.scalar_product(vector) < 0.0)
{
    return false;
}

Vector<T, 3> vectorV = cross_product(p1 - p3, intersection - p3);
if (normal.scalar_product(vectorV) < 0.0)
{
    return false;
}

T squareArea = normal.square_of_length();

v = sqrt(vectorV.square_of_length() / squareArea);
u = sqrt(vector.square_of_length() / squareArea);
return true;
}

```

Die Änderungen beziehen sich insbesondere auf die Berechnung von u und v . Diese wurden ans Ende der Funktion verschoben, da die Werte nur benötigt werden, wenn ein Schnittpunkt existiert. Zur Reduktion der Wurzelfunktion wurde die Berechnung von u und v angepasst: Anstatt die Länge durch die Fläche zu teilen, wird das Quadrat der Länge durch das Quadrat der Fläche geteilt und im Anschluss die Wurzel gezogen. Da bei der Berechnung der Länge und der Fläche jeweils die Wurzel aus dem jeweiligen Quadrat gezogen wird, kann so die Anzahl der `sqrt` Operationen von 3 (*2Länge + Fläche*) auf 2 ($2(\text{QuadratischeLänge}/\text{QuadratischeFläche})$) reduziert werden. Außerdem wird durch Prüfung, ob bereits ein näherer Schnittpunkt gefunden wird, ein früherer Abbruch der Funktion ermöglicht.

Assembler

Der Assembler-Code wurde ohne Compiler-Optimierung erzeugt, um die Lesbarkeit zu verbessern. Für die Ermöglichung des früheren Abbruchs (Überprüfung auf $t > \text{minimum_t}$) werden mehr Assembler-Befehle erzeugt. Der zweite Vergleich erfordert neben einer Vergleichsoperation (`VCOMISS`) auch einen zusätzlichen Sprungbefehl (`ja` und `jbe` im Vergleich zu nur `jbe`).

unoptimiert:

Listing 2: Assembler `minimum_t` unoptimiert

```

if (t < 0.0)
239: 48 8b 85 30 ff ff ff mov -0xd0(%rbp),%rax
240: c5 fa 10 08 vmovss (%rax),%xmm1
244: c5 f8 57 c0 vxorps %xmm0,%xmm0,%xmm0
248: c5 f8 2f c1 vcomiss %xmm1,%xmm0

```

```

24c: 76 0a jbe 258 <\
    ↳ _ZN8TriangleIfE10intersectsE6VectorIfLm3EES2_RfS3_S3_f+0
    ↳ x258>
{
return false;
24e: b8 00 00 00 00 mov $0x0,%eax
253: e9 1b 04 00 00 jmpq 673 <\
    ↳ _ZN8TriangleIfE10intersectsE6VectorIfLm3EES2_RfS3_S3_f+0
    ↳ x673>
}

```

optimiert:

Listing 3: Assembler minimum_t optimiert

```

if (t < 0.0 || t > minimum_t)
229: 48 8b 85 20 ff ff ff mov -0xe0(%rbp),%rax
230: c5 fa 10 08 vmovss (%rax),%xmm1
234: c5 f8 57 c0 vxorps %xmm0,%xmm0,%xmm0
238: c5 f8 2f c1 vcomiss %xmm1,%xmm0
23c: 77 15 ja 253 <\
    ↳ _ZN8TriangleIfE10intersectsE6VectorIfLm3EES2_RfS3_S3_f+0
    ↳ x253>
23e: 48 8b 85 20 ff ff ff mov -0xe0(%rbp),%rax
245: c5 fa 10 00 vmovss (%rax),%xmm0
249: c5 f8 2f 85 0c ff ff vcomiss -0xf4(%rbp),%xmm0
250: ff
251: 76 0a jbe 25d <\
    ↳ _ZN8TriangleIfE10intersectsE6VectorIfLm3EES2_RfS3_S3_f+0
    ↳ x25d>
{
return false;
253: b8 00 00 00 00 mov $0x0,%eax
258: e9 7a 04 00 00 jmpq 6d7 <\
    ↳ _ZN8TriangleIfE10intersectsE6VectorIfLm3EES2_RfS3_S3_f+0
    ↳ x6d7>
}

```

Die Optimierung der Berechnung von u und v führt zur Reduktion der Aufrufe der `sqrt` Implementierung der GLIBC. In der unoptimierten Variante wird die Fläche `area` als Länge des Normalenvektors berechnet:

Listing 4: Assembler Berechnung area

```

T area = normal.length(); // used for u-v-parameter calculation
6392: 48 8d 85 7c ff ff ff lea -0x84(%rbp),%rax
6399: 48 89 c7 mov %rax,%rdi

```

```

639c: e8 ed fa ff ff callq 5e8e <\_ZNK6VectorIfLm3EE6lengthEv>
63a1: c5 f9 7e c0 vmovd %xmm0,%eax
63a5: 89 45 f8 mov %eax,-0x8(%rbp)

```

Die Berechnung der Länge erfolgt in der Funktion `Vector::length`:

Listing 5: Assembler `Vector::length`

```

T length(void) const {
5e8e: 55 push %rbp
5e8f: 48 89 e5 mov %rsp,%rbp
5e92: 48 83 ec 10 sub $0x10,%rsp
5e96: 48 89 7d f8 mov %rdi,-0x8(%rbp)
return sqrt( square_of_length() );
5e9a: 48 8b 45 f8 mov -0x8(%rbp),%rax
5e9e: 48 89 c7 mov %rax,%rdi
5ea1: e8 8a 11 00 00 callq 7030 <\
    ↪ _ZNK6VectorIfLm3EE16square_of_lengthEv>
5ea6: c5 f2 5a c8 vcvts2sd %xmm0,%xmm1,%xmm1
5eaa: c4 e1 f9 7e c8 vmovq %xmm1,%rax
5eaf: c4 e1 f9 6e c0 vmovq %rax,%xmm0
5eb4: e8 47 c2 ff ff callq 2100 <sqrt@plt>
5eb9: c5 fb 5a c0 vcvtsd2ss %xmm0,%xmm0,%xmm0
}

```

Dazu wird zunächst `square_of_length()` aufgerufen (5ea1) und mit dem Ergebnis die GLIBC Implementierung der `sqrt` Funktion aufgerufen (5eb4). Auch für die Berechnung von `u` und `v` wird `Vector::length()` aufgerufen:

Listing 6: Assembler Berechnung `u` und `v`, unoptimiert

```

u = vector.length() / area;
6726: 48 8d 85 64 ff ff ff lea -0x9c(%rbp),%rax
672d: 48 89 c7 mov %rax,%rdi
6730: e8 59 f7 ff ff callq 5e8e <\_ZNK6VectorIfLm3EE6lengthEv>
6735: c5 fa 5e 45 f8 vdivss -0x8(%rbp),%xmm0,%xmm0
673a: 48 8b 85 28 ff ff ff mov -0xd8(%rbp),%rax
6741: c5 fa 11 00 vmovss %xmm0,(%rax)
...
v = vector.length() / area;
685b: 48 8d 85 64 ff ff ff lea -0x9c(%rbp),%rax
6862: 48 89 c7 mov %rax,%rdi
6865: e8 24 f6 ff ff callq 5e8e <\_ZNK6VectorIfLm3EE6lengthEv>
686a: c5 fa 5e 45 f8 vdivss -0x8(%rbp),%xmm0,%xmm0
686f: 48 8b 85 20 ff ff ff mov -0xe0(%rbp),%rax
6876: c5 fa 11 00 vmovss %xmm0,(%rax)

```

Insgesamt macht das 3 Aufrufe von `sqrt` (67830 und 6865 über `Vector::length`). Die optimierte Version verwendet die `Vector::length` Funktion nicht. Da für die Berechnung von `u` und `v` standardmäßig jeweils zwei Quadratwurzeln dividiert werden, kann die Wurzelberechnung auch auf das Ergebnis der Division erfolgen und damit verzögert werden. In der optimierten Version werden deshalb nur 2 Aufrufe von `sqrt` benötigt (6876 und 68ac).

Listing 7: Assembler Berechnung `u` und `v`, optimiert

```
v = sqrt(vectorV.square_of_length() / squareArea);
6854: 48 8d 85 58 ff ff ff lea -0xa8(%rbp),%rax
685b: 48 89 c7 mov %rax,%rdi
685e: e8 13 08 00 00 callq 7076 <\
    ↪ _ZNK6VectorIfLm3EE16square_of_lengthEv>
6863: c5 fa 5e 45 f4 vdivss -0xc(%rbp),%xmm0,%xmm0
6868: c5 d2 5a e8 vcvtsd2ss %xmm0,%xmm5,%xmm5
686c: c4 e1 f9 7e e8 vmovq %xmm5,%rax
6871: c4 e1 f9 6e c0 vmovq %rax,%xmm0
6876: e8 85 b8 ff ff callq 2100 <sqrt@plt>
687b: c5 fb 5a c0 vcvtsd2ss %xmm0,%xmm0,%xmm0
687f: 48 8b 85 10 ff ff ff mov -0xf0(%rbp),%rax
6886: c5 fa 11 00 vmovss %xmm0,(%rax)
u = sqrt(vector.square_of_length() / squareArea);
688a: 48 8d 85 64 ff ff ff lea -0x9c(%rbp),%rax
6891: 48 89 c7 mov %rax,%rdi
6894: e8 dd 07 00 00 callq 7076 <\
    ↪ _ZNK6VectorIfLm3EE16square_of_lengthEv>
6899: c5 fa 5e 45 f4 vdivss -0xc(%rbp),%xmm0,%xmm0
689e: c5 ca 5a f0 vcvtsd2ss %xmm0,%xmm6,%xmm6
68a2: c4 e1 f9 7e f0 vmovq %xmm6,%rax
68a7: c4 e1 f9 6e c0 vmovq %rax,%xmm0
68ac: e8 4f b8 ff ff callq 2100 <sqrt@plt>
68b1: c5 fb 5a c0 vcvtsd2ss %xmm0,%xmm0,%xmm0
68b5: 48 8b 85 18 ff ff ff mov -0xe8(%rbp),%rax
68bc: c5 fa 11 00 vmovss %xmm0,(%rax)
```

Messungen

Kompiliert wurde jeweils unter Debian 10 mit GCC 8.3.0. Die Messungen wurden auf einem PC mit folgenden Merkmalen erstellt:

- * CPU: AMD Ryzen 7 5800X @ 4.60 GHz
- * RAM: 32GB DDR4-3200

Messung ohne Optimierung

Durchlauf	Zeit
1	4.62049 s
2	4.60774 s
3	4.56693 s
4	4.56833 s
5	4.58256 s
6	4.58978 s
7	4.56290 s
8	4.57446 s
9	4.55789 s
10	4.59217 s
Durchschnitt	4.582325 s

Messung mit Optimierung

Durchlauf	Zeit
1	4.02156 s
2	4.07757 s
3	4.06602 s
4	4.04903 s
5	4.07341 s
6	4.0889 s
7	4.09844 s
8	4.09043 s
9	4.08588 s
10	4.0745 s
Durchschnitt	4.072574 s

Ergibt eine Performance-Steigerung von etwa +12,5 %. Zurückzuführen ist diese Steigerung auf die Reduktion der Quadratwurzelberechnung. Durch die Ausführung der Berechnungen am Ende der Funktion werden Quadratwurzeln nur noch berechnet wenn dies notwendig ist. Zusätzlich wurde die maximale Anzahl der Quadratwurzelberechnungen reduziert und durch die Prüfung, ob bereits nähere Schnittpunkte gefunden wurden, kann die Berechnung häufiger frühzeitig abgebrochen werden.

2. Quadratwurzelberechnung optimieren

Optimierung der Quadratwurzelberechnung durch Verwendung des Newton-Verfahrens. Wo möglich wird der Assembler-Code ohne Compileroptimierung verwendet, um die Lesbarkeit zu erhöhen. Sourcecode sqrt1:

Listing 8: sqrt1

```
template <size_t LOOPS = 2>
float sqrt1(float *a)
{
    float root;

    int *ai = reinterpret_cast<int *>(a);
    int *initial = reinterpret_cast<int *>(&root);
    *initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;
    root = *reinterpret_cast<float *>(initial);
    // newton method
    for (size_t i = 0; i < LOOPS; i++)
    {
        root = 0.5 * (root + *a / root);
    }

    return root;
}
```

Der Assembler-Code für sqrt1 ohne Compiler-Optimierung:

Listing 9: sqrt1 Assembler

```
template <size_t LOOPS = 2>
float sqrt1(float *a)
    18ef: 55 push %rbp
    18f0: 48 89 e5 mov %rsp,%rbp
    18f3: 48 89 7d d8 mov %rdi,-0x28(%rbp)
{
    float root;

    int *ai = reinterpret_cast<int *>(a);
    18f7: 48 8b 45 d8 mov -0x28(%rbp),%rax
    18fb: 48 89 45 f0 mov %rax,-0x10(%rbp)
    int *initial = reinterpret_cast<int *>(&root);
    18ff: 48 8d 45 e4 lea -0x1c(%rbp),%rax
    1903: 48 89 45 e8 mov %rax,-0x18(%rbp)
    *initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;
    1907: 48 8b 45 f0 mov -0x10(%rbp),%rax
    190b: 8b 00 mov (%rax),%eax
    190d: d1 f8 sar %eax
```

```

190f: 8d 90 00 40 bb 1f lea 0x1fbb4000(%rax),%edx
1915: 48 8b 45 e8 mov -0x18(%rbp),%rax
1919: 89 10 mov %edx,(%rax)
root = *reinterpret_cast<float *>(initial);
191b: 48 8b 45 e8 mov -0x18(%rbp),%rax
191f: c5 fa 10 00 vmovss (%rax),%xmm0
1923: c5 fa 11 45 vmovss %xmm0,-0x1c(%rbp)
// newton method
for (size_t i = 0; i < LOOPS; i++)
1928: 48 c7 45 f8 00 00 00 movq $0x0,-0x8(%rbp)
192f: 00
1930: 48 83 7d f8 01 cmpq $0x1,-0x8(%rbp)
1935: 77 31 ja 1968 <_Z5sqrt1ILm2EEfPf+0x79>
{
    root = 0.5 * (root + *a / root);
1937: 48 8b 45 d8 mov -0x28(%rbp),%rax
193b: c5 fa 10 00 vmovss (%rax),%xmm0
193f: c5 fa 10 4d e4 vmovss -0x1c(%rbp),%xmm1
1944: c5 fa 5e c1 vdivss %xmm1,%xmm0,%xmm0
1948: c5 fa 10 4d e4 vmovss -0x1c(%rbp),%xmm1
194d: c5 fa 58 c1 vaddss %xmm1,%xmm0,%xmm0
1951: c5 fa 10 0d 0f 38 00 vmovss 0x380f(%rip),%xmm1 # 5168 <
    ↪ _ZStL19piecewise_construct+0x158>
1958: 00
1959: c5 fa 59 c1 vmulss %xmm1,%xmm0,%xmm0
195d: c5 fa 11 45 e4 vmovss %xmm0,-0x1c(%rbp)
for (size_t i = 0; i < LOOPS; i++)
1962: 48 ff 45 f8 incq -0x8(%rbp)
1966: eb c8 jmp 1930 <_Z5sqrt1ILm2EEfPf+0x41>
}

return root;
1968: c5 fa 10 45 e4 vmovss -0x1c(%rbp),%xmm0
}

```

Die zweite Variante berechnet vier Wurzeln gleichzeitig. Der Compiler erzeugt dazu automatisch die Packed-SIMD Instruktionen.

Listing 10: sqrt2

```

template <size_t LOOPS = 2>
void sqrt2(float *__restrict__ a, float *__restrict__ root)
{
    int *ai = reinterpret_cast<int *>(a);
    int *initial = reinterpret_cast<int *>(root);
    initial[0] = (1 << 29) + (ai[0] >> 1) - (1 << 22) - 0x4C000;
    initial[1] = (1 << 29) + (ai[1] >> 1) - (1 << 22) - 0x4C000;
}

```



```

initial[2] = (1 << 29) + (ai[2] >> 1) - (1 << 22) - 0x4C000;
initial[3] = (1 << 29) + (ai[3] >> 1) - (1 << 22) - 0x4C000;
root = reinterpret_cast<float *>(initial);
// newton method
for (size_t i = 0; i < LOOPS; i++)
{
    root[0] = 0.5 * (root[0] + a[0] / root[0]);
    root[1] = 0.5 * (root[1] + a[1] / root[1]);
    root[2] = 0.5 * (root[2] + a[2] / root[2]);
    root[3] = 0.5 * (root[3] + a[3] / root[3]);
}
}

```

Der Assembler-Code für sqrt2 wurde mit Optimierung (-O3) erzeugt, damit die Packed-SIMD Instruktionen verwendet werden. Durch Optimierungen wie Inlining wird der Assembler-Code allerdings schwer lesbar und Anfang und Ende einer Funktion sind nur schwer identifizierbar. Um dennoch lesbaren Assembler-Code zu erhalten wurde die Main-Funktion angepasst, sodass nur noch die Funktion sqrt2 mit zufälligen Zahlen mehrfach aufgerufen wird.

Listing 11: Assembler für sqrt2

```

template <size_t LOOPS = 2>
void sqrt2(float *__restrict__ a, float *__restrict__ root)
{
    int *ai = reinterpret_cast<int *>(a);
    int *initial = reinterpret_cast<int *>(root);
    initial[0] = (1 << 29) + (ai[0] >> 1) - (1 << 22) - 0x4C000;
    1190: c4 c1 78 28 44 24 20 vmovaps 0x20(%r12),%xmm0
    initial[3] = (1 << 29) + (ai[3] >> 1) - (1 << 22) - 0x4C000;
    root = reinterpret_cast<float *>(initial);
    // newton method
    for (size_t i = 0; i < LOOPS; i++)
    {
        root[0] = 0.5 * (root[0] + a[0] / root[0]);
        1197: c5 f8 28 60 20 vmovaps 0x20(%rax),%xmm4
        119c: 48 83 c0 40 add $0x40,%rax
        11a0: 49 83 c4 40 add $0x40,%r12
        initial[0] = (1 << 29) + (ai[0] >> 1) - (1 << 22) - 0x4C000;
        11a4: c4 c1 78 28 4c 24 c0 vmovaps -0x40(%r12),%xmm1
        11ab: c4 41 78 c6 54 24 f0 vshufps $0x88,-0x10(%r12),%xmm0,%
            ↪ xmm10
        11b2: 88
        11b3: c4 41 78 c6 44 24 f0 vshufps $0xdd,-0x10(%r12),%xmm0,%
            ↪ xmm8
        11ba: dd
        root[0] = 0.5 * (root[0] + a[0] / root[0]);
    }
}

```

```

11bb: c5 f8 28 40 c0 vmovaps -0x40(%rax),%xmm0
11c0: c5 d8 c6 50 f0 88 vshufps $0x88,-0x10(%rax),%xmm4,%xmm2
initial[0] = (1 << 29) + (ai[0] >> 1) - (1 << 22) - 0x4C000;
11c6: c4 c1 70 c6 5c 24 d0 vshufps $0x88,-0x30(%r12),%xmm1,%
    ↪ xmm3
11cd: 88
root[0] = 0.5 * (root[0] + a[0] / root[0]);
11ce: c5 d8 c6 60 f0 dd vshufps $0xdd,-0x10(%rax),%xmm4,%xmm4
initial[0] = (1 << 29) + (ai[0] >> 1) - (1 << 22) - 0x4C000;
11d4: c4 c1 70 c6 4c 24 d0 vshufps $0xdd,-0x30(%r12),%xmm1,%
    ↪ xmm1
11db: dd
root[0] = 0.5 * (root[0] + a[0] / root[0]);
11dc: c5 78 c6 48 d0 88 vshufps $0x88,-0x30(%rax),%xmm0,%xmm9
11e2: c5 f8 c6 40 d0 dd vshufps $0xdd,-0x30(%rax),%xmm0,%xmm0
initial[0] = (1 << 29) + (ai[0] >> 1) - (1 << 22) - 0x4C000;
11e8: c4 c1 60 c6 ea 88 vshufps $0x88,%xmm10,%xmm3,%xmm5
root[0] = 0.5 * (root[0] + a[0] / root[0]);
11ee: c5 f8 c6 fc 88 vshufps $0x88,%xmm4,%xmm0,%xmm7
11f3: c5 30 c6 da 88 vshufps $0x88,%xmm2,%xmm9,%xmm11
11f8: c5 f8 c6 e4 dd vshufps $0xdd,%xmm4,%xmm0,%xmm4
initial[0] = (1 << 29) + (ai[0] >> 1) - (1 << 22) - 0x4C000;
11fd: c4 c1 70 c6 c0 88 vshufps $0x88,%xmm8,%xmm1,%xmm0
root[0] = 0.5 * (root[0] + a[0] / root[0]);
1203: c4 c1 50 5e eb vdivps %xmm11,%xmm5,%xmm5
1208: c5 30 c6 ca dd vshufps $0xdd,%xmm2,%xmm9,%xmm9
root[1] = 0.5 * (root[1] + a[1] / root[1]);
120d: c5 f8 5e c7 vdivps %xmm7,%xmm0,%xmm0
root[0] = 0.5 * (root[0] + a[0] / root[0]);
1211: c4 c1 50 58 d3 vaddps %xmm11,%xmm5,%xmm2
1216: c5 e8 59 ee vmulps %xmm6,%xmm2,%xmm5
initial[0] = (1 << 29) + (ai[0] >> 1) - (1 << 22) - 0x4C000;
121a: c4 c1 60 c6 d2 dd vshufps $0xdd,%xmm10,%xmm3,%xmm2
root[1] = 0.5 * (root[1] + a[1] / root[1]);
1220: c5 f8 58 c7 vaddps %xmm7,%xmm0,%xmm0
root[2] = 0.5 * (root[2] + a[2] / root[2]);
1224: c4 c1 68 5e d1 vdivps %xmm9,%xmm2,%xmm2
root[1] = 0.5 * (root[1] + a[1] / root[1]);
1229: c5 f8 59 fe vmulps %xmm6,%xmm0,%xmm7
initial[0] = (1 << 29) + (ai[0] >> 1) - (1 << 22) - 0x4C000;
122d: c4 c1 70 c6 c0 dd vshufps $0xdd,%xmm8,%xmm1,%xmm0
root[3] = 0.5 * (root[3] + a[3] / root[3]);
1233: c5 f8 5e c4 vdivps %xmm4,%xmm0,%xmm0
root[2] = 0.5 * (root[2] + a[2] / root[2]);
1237: c4 c1 68 58 d1 vaddps %xmm9,%xmm2,%xmm2
123c: c5 e8 59 d6 vmulps %xmm6,%xmm2,%xmm2

```

```

root[3] = 0.5 * (root[3] + a[3] / root[3]);
1240: c5 d0 14 ca vunpcklps %xmm2,%xmm5,%xmm1
1244: c5 d0 15 d2 vunpckhps %xmm2,%xmm5,%xmm2
1248: c5 f8 58 c4 vaddps %xmm4,%xmm0,%xmm0
124c: c5 f8 59 c6 vmulps %xmm6,%xmm0,%xmm0
1250: c5 c0 14 d8 vunpcklps %xmm0,%xmm7,%xmm3
1254: c5 c0 15 c0 vunpckhps %xmm0,%xmm7,%xmm0
1258: c5 f0 14 e3 vunpcklps %xmm3,%xmm1,%xmm4
125c: c5 f0 15 cb vunpckhps %xmm3,%xmm1,%xmm1
1260: c5 f8 29 48 d0 vmovaps %xmm1,-0x30(%rax)
1265: c5 e8 14 c8 vunpcklps %xmm0,%xmm2,%xmm1
1269: c5 e8 15 d0 vunpckhps %xmm0,%xmm2,%xmm2
126d: c5 f8 29 60 c0 vmovaps %xmm4,-0x40(%rax)
1272: c5 f8 29 48 e0 vmovaps %xmm1,-0x20(%rax)
1277: c5 f8 29 50 f0 vmovaps %xmm2,-0x10(%rax)
127c: 48 39 c2 cmp %rax,%rdx
127f: 0f 85 0b ff ff ff jne 1190 <main+0x70>

```

Im Assembler-Code fällt auf, dass die Packed-SIMD Instruktionen `vdivps`, `vaddps` und `vmulps` für die Berechnung der Wurzeln verwendet werden. Außerdem ist zu sehen, dass der Compiler ein Loop-Unrolling durchgeführt hat. #

Für die dritte Variante werden Packed-SIMD Instruktionen genutzt um 4 Werte gleichzeitig zu berechnen. Sourcecode `v4sf_sqrt`:

Listing 12: `v4sf_sqrt`

```

template <size_t LOOPS = 2>
float v4sf_sqrt(float *a)
{
    v4si *ai = reinterpret_cast<v4si *>(a);
    v4si *initial = reinterpret_cast<v4si *>(root);
    *initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;

    root = reinterpret_cast<v4sf *>(initial);
    // newton method
    for (size_t i = 0; i < LOOPS; i++)
    {
        *root = 0.5 * (*root + *a / *root);
    }
}

```

Durch die Verwendung der Compiler-Intrinsics, verwendet der Compiler die Packed-SIMD Instruktionen auch ohne Compiler-Optimierung. Deshalb kann hier der unoptimierte, besser lesbare Assembler verwendet werden.

Listing 13: `v4sf_sqrt` Assembler

```

void v4sf_sqrt(v4sf *__restrict__ a, v4sf *__restrict__ root)
    22fb: 55 push %rbp
    22fc: 48 89 e5 mov %rsp,%rbp
    22ff: 48 89 7d d8 mov %rdi,-0x28(%rbp)
    2303: 48 89 75 d0 mov %rsi,-0x30(%rbp)
    v4si *ai = reinterpret_cast<v4si *>(a);
    2307: 48 8b 45 d8 mov -0x28(%rbp),%rax
    230b: 48 89 45 f0 mov %rax,-0x10(%rbp)
    v4si *initial = reinterpret_cast<v4si *>(root);
    230f: 48 8b 45 d0 mov -0x30(%rbp),%rax
    2313: 48 89 45 e8 mov %rax,-0x18(%rbp)
    *initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;
    2317: 48 8b 45 f0 mov -0x10(%rbp),%rax
    231b: c5 f8 28 00 vmovaps (%rax),%xmm0
    231f: c5 f1 72 e0 01 vpsrad $0x1,%xmm0,%xmm1
    2324: c5 f8 28 05 24 0e 00 vmovaps 0xe24(%rip),%xmm0 # 3150 <
        ↪ _ZStL19piecewise_construct+0x140>
    232b: 00
    232c: c5 f1 fe c0 vpaddd %xmm0,%xmm1,%xmm0
    2330: 48 8b 45 e8 mov -0x18(%rbp),%rax
    2334: c5 f8 29 00 vmovaps %xmm0,(%rax)
    root = reinterpret_cast<v4sf *>(initial);
    2338: 48 8b 45 e8 mov -0x18(%rbp),%rax
    233c: 48 89 45 d0 mov %rax,-0x30(%rbp)
    for (size_t i = 0; i < LOOPS; i++)
    2340: 48 c7 45 f8 00 00 00 movq $0x0,-0x8(%rbp)
    2347: 00
    2348: 48 83 7d f8 01 cmpq $0x1,-0x8(%rbp)
    234d: 77 3a ja 2389 <_Z9v4sf_sqrtILm2EEvPDv4_fS1_+0x8e>
    *root = 0.5 * (*root + *a / *root);
    234f: 48 8b 45 d0 mov -0x30(%rbp),%rax
    2353: c5 f8 28 08 vmovaps (%rax),%xmm1
    2357: 48 8b 45 d8 mov -0x28(%rbp),%rax
    235b: c5 f8 28 00 vmovaps (%rax),%xmm0
    235f: 48 8b 45 d0 mov -0x30(%rbp),%rax
    2363: c5 f8 28 10 vmovaps (%rax),%xmm2
    2367: c5 f8 5e c2 vdivps %xmm2,%xmm0,%xmm0
    236b: c5 f0 58 c8 vaddps %xmm0,%xmm1,%xmm1
    236f: c5 f8 28 05 e9 0d 00 vmovaps 0xde9(%rip),%xmm0 # 3160 <
        ↪ _ZStL19piecewise_construct+0x150>
    2376: 00
    2377: c5 f0 59 c0 vmulps %xmm0,%xmm1,%xmm0
    237b: 48 8b 45 d0 mov -0x30(%rbp),%rax
    237f: c5 f8 29 00 vmovaps %xmm0,(%rax)
    for (size_t i = 0; i < LOOPS; i++)
    2383: 48 ff 45 f8 incq -0x8(%rbp)

```

```

2387: eb bf jmp 2348 <_Z9v4sf_sqrtILm2EEvPDv4_fS1_+0x4d>
}
2389: 90 nop
238a: 5d pop %rbp
238b: c3 retq

```

Messungen

Kompiliert wurde jeweils unter Debian 10 mit GCC 7.4.0. Die GCC Version 8.3.0 lieferte durch zusätzliche Optimierungen für sqrt3 Zeiten von unter 1000 ns, was im Vergleich mit den anderen Zeiten unrealistisch erscheint, eventuell wurden Optimierungen durchgeführt, welche die Zeitmessung beeinflussen. Die Messungen wurden auf einem PC mit folgenden Merkmalen erstellt:

* CPU: AMD Ryzen 7 5800X @ 4.60 GHz

* RAM: 32GB DDR4-3200

2 Iterationen

Durchlauf	math.sqrt [ns]	sqrt1 [ns]	sqrt1*4 [ns]	sqrt2 [ns]	sqrt3 [ns]
1	1244731	132099	159254	157880	132355
2	1232933	131793	158757	158193	131847
3	1244356	131728	158531	158055	131932
4	1238731	132128	158533	157230	131758
5	1241204	132227	158571	157943	134218
6	1240188	132393	158445	157378	131885
7	1253477	132444	158934	157814	137585
8	1240861	133277	161816	159400	132265
9	1243250	131988	158477	157781	132435
10	1244608	132120	159346	157520	132092
—	—	—	—	—	—
Durchschnitt	1242433.9	132219.7	159066.4	157919.4	132837.2

3 Iterationen

Durchlauf	math.sqrt [ns]	sqrt1 [ns]	sqrt1*4 [ns]	sqrt2 [ns]	sqrt3 [ns]
1	1274224	199425	231799	235836	200774
2	1241694	197401	230817	235878	198245
3	1244044	197625	231488	234478	198096
4	1241951	197603	230895	234020	197587
5	1243158	199289	231197	235173	198257
6	1242758	200338	238524	234554	201696

Durchlauf	math.sqrt [ns]	sqrt1 [ns]	sqrt1*4 [ns]	sqrt2 [ns]	sqrt3 [ns]
7	1251411	197677	231318	234644	198069
8	1240883	197868	231052	233187	198576
9	1246030	198530	231148	233217	198022
10	1242813	197660	231288	233682	197976
Durchschnitt	1246896.6	198341.6	231952.6	234466.9	198729.8

4 Iterationen

Durchlauf	math.sqrt [ns]	sqrt1 [ns]	sqrt1*4 [ns]	sqrt2 [ns]	sqrt3 [ns]
1	1250898	270484	319153	319681	269713
2	1242425	267495	316088	317749	267528
3	1245181	268141	316043	317910	267465
4	1244542	268034	316127	317757	271406
5	1243158	199289	231197	235173	198257
6	1254091	268371	316607	333658	274396
7	1246632	267236	317179	318522	268319
8	1244199	267383	315701	318046	267868
9	1242659	267807	315654	317933	269594
10	1243171	267500	315900	318000	267500
Durchschnitt	1245695.6	261174	307964.9	311442.9	262204.6

Wie erwartet, ist jede der optimierten Versionen schneller als die Standard-Implementierung der GLIBC. Ebenfalls wie erwartet, steigt die Ausführungszeit mit der Anzahl der Anzahl Iterationen.

Die Variante `sqrt1` mit nur zwei Iterationen ist am schnellsten, gefolgt von der Variante `sqrt3` welche SIMD-Befehle nutzt um die Berechnung mehrerer Quadratwurzeln zu beschleunigen. Die Variante `sqrt2` ist etwas langsamer als `sqrt3`, was vermutlich auf schlechtere Optimierung durch den Compiler bei Verzicht auf die Nutzung der SIMD-Intrinsics zurückzuführen ist. Die Automatische Vektorisierung durch den Compiler scheint schlechter zu funktionieren als die manuelle Optimierung durch die Verwendung von SIMD-Intrinsics. Allerdings sind die Unterschiede vergleichsweise klein. Dass `sqrt1` mit der zusätzlichen inneren Schleife langsamer ist als `sqrt1`, lässt sich darauf zurückführen, dass der Compiler die verschachtelten Schleifen weniger gut optimieren kann als die einzelne Schleife. Erwartet wurde, dass die Vektorisierte Variante `sqrt3` schneller ist als die Variante `sqrt1`, da die SIMD-Instruktionen die Berechnung von mehreren Quadratwurzeln gleichzeitig ermöglichen. Jedoch ist die Quadratwurzelberechnung vergleichsweise einfach, sodass ein großer Teil

der Berechnungszeit auf Speicherbefehle (Load/Store) entfällt und die SIMD-Instruktionen ihr volles Potenzial nicht ausschöpfen können.