Square and Cube Root Optimization

Team 25:
Keegan Kavanagh
Jon Squire

Tasks

1. Compute the calculation of square roots and cube roots
2. Optimize the resulting algorithm of for the ARM machine
3. Analyze how different optimizations affect the square root and cube root function differently

Requirements

- Compiled using gcc version 4.3.2 for the 2.6.29.4-FriendlyARM platform
- Written in C
- Square root input range of [1,4)
- Cube root input range of [1,8)
- 32 bit output

The CCM Algorithm for Computing Roots

The Convergence Computing method for calculating the roots of number reduces the process down into two key instructions

- Bit shifts
- Additions

HOWEVER! Not only is floating point arithmetic computationally taxing, it is also not possible to shift floating point values.

Pseudo Code:

Assuming:
M = input
K = bits of precision

```
f = 1.0
f_sqrt = 1.0
for i = 0 to K −1 do
        μ = f ·(1+2−i ) ·(1+2^(−i) )
        μ_sqrt = f_sqrt ·(1+2^(−i) )
        if μ <= M then
                f = μ
                f_sqrt = μ_sqrt
return f_sqrt
```

Software solution: Fixed Point Arithmetic

1. Introduce a scale factor to the floating point input and algorithm
2. Convert input into 32 bit integer
3. Perform calculation
4. Convert integer output back into type float
5. Remove the scale factor from the output

37.75 -> 4.22s !

```
[jonathonsquire@seng440 src]$ ./qRoot.exe
The Square Root of 1.000 is 1.000000
The Square Root of 2.750 is 1.400940
The Square Root of 3.000 is 1.442200
The Square Root of 4.250 is 1.619751
The Square Root of 5.000 is 1.709900
The Square Root of 6.500 is 1.866089
The Square Root of 7.999 is 1.999512
```

Increased efficiency by 90%!

48.05 -> 4.86s !

```
[jonathonsquire@seng440 src]$ ./qRoot.exe
The Cube Root of 1.000 is 1.000000
The Cube Root of 2.750 is 1.400940
The Cube Root of 3.000 is 1.442200
The Cube Root of 4.250 is 1.619751
The Cube Root of 5.000 is 1.709900
The Cube Root of 6.500 is 1.866089
The Cube Root of 7.999 is 1.999512
```

Software solution: C Optimizations

Pipelining

```c
u = f+(f>>0);
u_sqrt =  f_sqrt + (f_sqrt>>0);
u = u+(u>>0);

for (i=1; i<16; i++){
    if (u <= local_M){
        f = u;
        f_sqrt = u_sqrt;
    }
    u = f + (f >> i);
    u_sqrt =  f_sqrt + (f_sqrt >> i);
    u = u + (u >> i);
}
if (u <= local_M){
        f = u;
        f_sqrt = u_sqrt;
}
```

Operator Reduction of Loops

```c
for(i=1;!(i&16);i++){
        ...
        if((u-local_M-1)&2147483648){
                ...
```

Operator Reduction of First Iteration

```c
register int32_t u = f<<2;
register int32_t u_sqrt =  f_sqrt<<1;
```

Loop Unrolling

```c
...
if (u <= local_M){
        f = u;
        f_sqrt = u_sqrt;
}
u = f + (f >> i);
u_sqrt =  f_sqrt + (f_sqrt >> i);
u = u + (u >> i);

if (u <= local_M){
        f = u;
        f_sqrt = u_sqrt;
}
i++;
u = f + (f >> i);
u_sqrt =  f_sqrt + (f_sqrt >> i);
u = u + (u >> i);
...
```

Software solution: Assembly optimizations

arm-linux-gcc -S ...

```
str   fp, [sp, #-4]!
add   fp, sp, #0
sub   sp, sp, #36
str   r0, [fp, #-8]
mov   r1, #16384
str   r1, [fp, #-32]
mov   r2, #16384
str   r2, [fp, #-28]
ldr   r3, [fp, #-8]
str   r3, [fp, #-24]
ldr   r1, [fp, #-32]
mov   r1, r1, asl #2
str   r1, [fp, #-16]
...
```

arm-linux-gcc -S -O3 ...

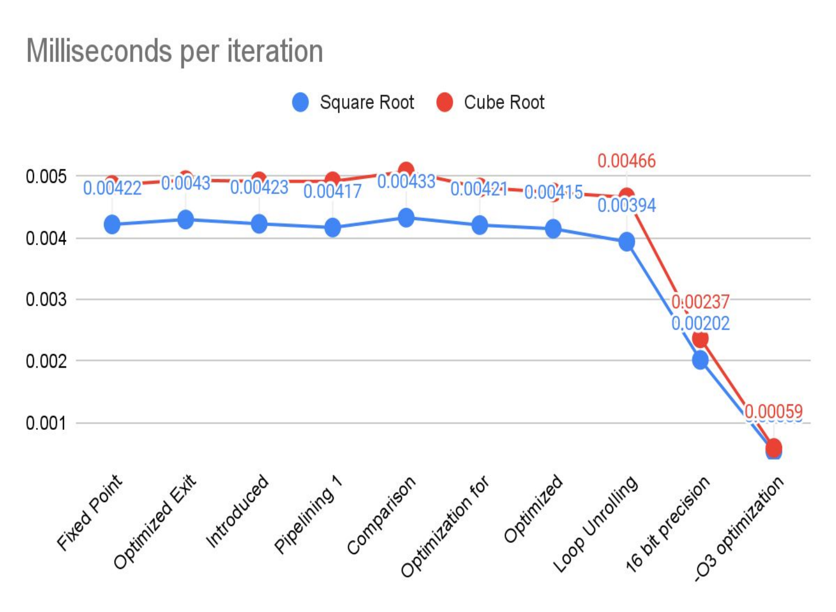
```
movle       r0, #16384
movgt       r0, #65536
add   r2, r0, r0, lsr #1
add   r2, r2, r2, lsr #1
movle       r3, #16384
movgt       r3, #32768
cmp   ip, r2
movge       r0, r2
add   r1, r0, r0, lsr #2
add   r1, r1, r1, lsr #2
add   r2, r3, r3, lsr #1
movge       r3, r2
...
```

Pigeonhole optimization

```
...
add   r3, r3, r3, lsr #1
...
```

Discussion

- Introducing predicates did not always result in faster runtimes
- Each optimizations impacted the different algorithms differently
- All of the load/str were replaced by add and move instructions
- Assembly file had gotten larger with optimization but was faster as the individual instructions were much less taxing



Vertical or Horizontal Machine?

Vertical Machine: only one pipeline stage is executed at any given time

Horizontal Machine: set number of pipeline stages can be executing simultaneously

Considering each iteration is dependant on the previous iterations computation, horizontal machines introduce unnecessary complexity, resulting in excessive NOP instructions.

Therefore: a vertical machine is sufficient for this project

Conclusion

- Took a lot of time to get fixed point working, but once working we optimized fast
- Had an increase of 8411% in performance in the cube root time

Questions?