Approximate Reciprocal Square Root with Single- and Half-Precision Floats

Matheus M. Susin, Lucas Francisco Wanner

University of Campinas

matheus.susin@students.ic.unicamp.br, lucas@ic.unicamp.br

Abstract. *In this work, we compared the precision, speed, and power consumption of the reciprocal square root of a single-precision floating point number, using different approximation techniques. We also devised an equivalent approximation for half-precision floating point numbers, and evaluated its performance across the whole range of positive non-zero 16-bit floating point values.*

1. Introduction

The reciprocal square root, $f(x) = \frac{1.0}{\sqrt{x}}$, is used to normalize vectors, an important part of calculating angles of incidence and reflection for lighting and shading. Before the introduction of Streaming SIMD Execution (SSE) by Intel in the Pentium III series, in 1999, home computers were slow at evaluating this function. 3D companies such as 3dfx and Silicon Graphics used an approximation technique that starts from a seemingly arbitrary starting point, performs one or more Newton-Raphson iterations, and achieves a relative error that is smaller than 2%. Given that a brute-force approach to the challenge of finding a good starting point in the mid-1980s would take a very long time, it is believed that the starting point was found by a bisection method [McEniry 2007].

Despite this technique having passed through several companies whose products required fast 3D computation, it only became widely known after the source code for Quake III Arena, by id Software, was made available. It has thus become known as the "Carmack Fast Inverse Square Root", after John Carmack, who implemented several tricks for fast computation at id [Kushner 2002].

2. Objectives

In this work, we intend to evaluate the accuracy of the original function for a given scenario. We also wish to measure whether it is faster and/or more energy-efficient than computing the function in hardware.

Finally, we wish to find an ideal constant to replicate the implementation of Carmack's Fast Inverse Square Root for half-precision floats [IEEE 754-2008 2008].

3. Code

```
1  float Q_rsqrt(float number)
2  {
3    int i;
4    float x2, y;
5    const float threehalfs = 1.5F;
6
7    x2 = number * 0.5F;
8    y  = number;
9    i  = *(int *)&y;
10   i  = 0x5f3759df - (i >> 1);
11   y  = *(float *)&i;
12   y  = y * (threehalfs - (x2 * y * y));  // 1st iteration
13   //  y  = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can be removed
14
15   return y;
16 }
```

Listing 1. Carmack's implementation of an approximation for the reciprocal square root

```
import random
import numpy as np
import itertools

iteration_size = 1048576
f = open("float32_dataset.txt", "w+")

float32_exp_range = range(-6, 7)
combinations = [pair for pair in itertools.combinations(float32_exp_range, 2)]
total_size = len(combinations) * iteration_size
f.write(str(total_size) + "\n")
i = 0
for pair in combinations:
    for k in range(iteration_size):
        uniform_start = float("1.0e" + str(pair[0]))
        uniform_end = float("1.0e" + str(pair[1]))
        x = np.float32(random.uniform(uniform_start, uniform_end))
        y = np.float32(random.uniform(uniform_start, uniform_end))
        z = np.float32(random.uniform(uniform_start, uniform_end))
        v = x * x + y * y + z * z
        f.write(("%e" % v) + "\n")
        i += 1
        # print(str(i) + " / " + str(total_size))
```

Listing 2. Python script that generates over 81 million FP32 values

4. Methods

We evaluate the performance and precision of Carmack's implementation of the function, comparing it to `1.0 / sqrtf(x)`, using `math.h`. We perform these steps with and without the `-ffast-math` flag. In total, there are 4 different compilation outputs, each a combination of these settings. We also estimate energy consumption using Intel RAPL counters accessed via `perf`. All experiments were run 2500 times.

To find the constant for half-precision floats, we iterate over all normal positive non-zero values of half-precision floats, trying out all possible 16-bit constants, and settling for the one with the smallest maximum error. Only the range of positive normal numbers was considered, that is, from `0x0040` to `0x7bff`.

We wrote a variant of Listing 1 that takes a half-precision floating point number, and a magic number to use as the constant in line 10. We listed all constants that, when passed with a normal positive half-precision float to the function, would yield a normal positive half-precision float. We then compared this result with casting the half-precision float to a single-precision float, calling `sqrtf`, then casting it back to half-precision float.

5. Results

Table 5 shows the execution time and energy consumption of the 4 binary files. They were all compiled with `-O3 -march=native -mtune=native` on an Intel Core i5-4590 3.30GHz CPU with gcc 6.3.0-18 for Debian.

The 4 programs are:

1. `no-approx`: Using `1.0 / sqrtf(x)`, and no additional compilation flags.
2. `fast-math`: Using `1.0 / sqrtf(x)`, and `-ffast-math` as an additional compilation flag.
3. `quake`: Using Carmack's implementation, and no additional compilation flags.
4. `quake-fast`: Using Carmack's implementation, and `-ffast-math` as an additional compilation flag.

PROGRAM	MAX RELATIVE ERROR	EXECUTION TIME	ENERGY
no-approx	N/A (used as baseline)	649.69 ± 002.34 ms	14.06 ± 00.22 J
fast-math	$3.0 \times 10^{-5}\%$	146.30 ± 001.03 ms	03.90 ± 00.06 J
quake	0.175%	090.86 ± 000.95 ms	02.92 ± 00.05 J
quake-fast	0.175%	090.06 ± 000.83 ms	02.90 ± 00.05 J

Table 1. Results obtained from running the 4 different programs

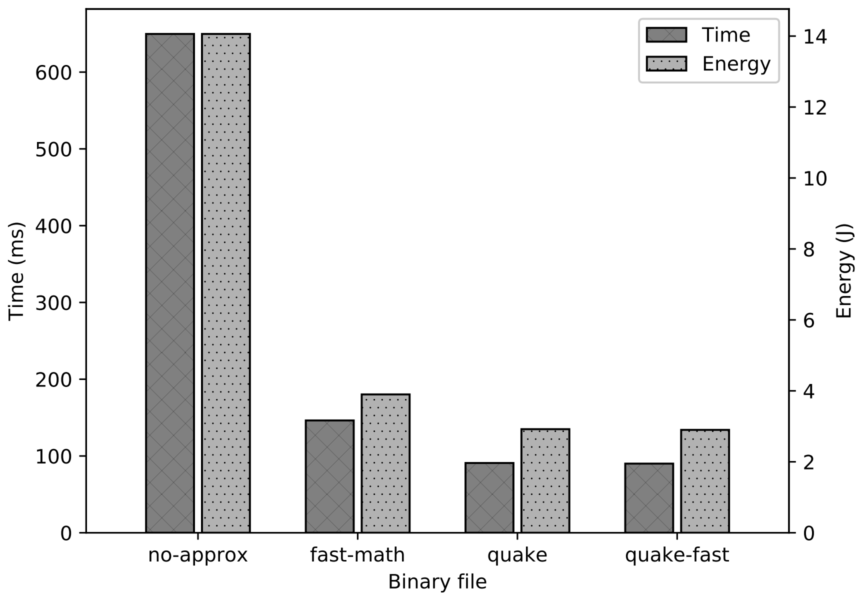

Figure 1. Bar chart comparing the 4 versions of the program in elapsed time and consumed energy

The 16-bit constant that yields the best result for an equivalent function over half-precision floats is `0x59b8`, with a maximum relative error of 0.20%. Due to the unavailability of hardware support for the floating-point-16 type in the CPUs that we had access to, a half-precision software library [Rau 1 16] was used, greatly increasing the overhead. We were thus unable to evaluate its performance.

6. Conclusion

Even in software, Carmack's approximation still performs better than Intel's `vrsqrtss` SSE instruction. If low latency is required, and thus offloading to an accelerator is not viable, this function can be used instead. It is 4 orders of magnitude less precise than the hardware alternative, which performs several Newton-Raphson steps, so the programmer must be aware of the application requirements before replacing one with the other. Providing several instructions in hardware with variable numbers of Newton-Raphson iterations could allow the developer to choose how much precision to use, trading off performance.

Moreover, we did not evaluate the entire range of 32-bit floating point numbers. Better or worse accuracy may be achieved over different ranges. Again, understanding the application is paramount.

Finally, we found an ideal constant for the entire range of normal positive half-precision floating point values. A different constant may be used if the range is limited, and the relative error needs to be smaller than 0.20%. For an equivalent double-precision constant, found via the bisection method, refer to [McEniry 2007].

Acknowledgements

This work was supported in part by CNPq, CAPES, and FAPESP.

References

IEEE 754-2008 (2008). IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*.

Kushner, D. (2002). The wizardry of id. *IEEE Spectrum*, pages 42–47.

McEniry, C. (2007). The mathematics behind the fast inverse square root function code.

Rau, C. (accessed on 2017-11-16). IEEE 754-based half-precision floating point library. `https://half.sourceforge.net/`.