Convergence Computing Method

Samantha Lai
University of Victoria Dept. of SEng
V00846643
slai@uvic.ca

Submission Date: _____

Table of Contents

Introduction

This project was constructed for the University of Victoria's SENG440 Embedded Systems course held in summer 2018. The convergence computing method (CCM) algorithm was provided by Professor Mihai Sima [1]. The program described here is designed and optimized for execution on a 32-bit ARM processor. Calculations are performed with 32-bits of precision under a scale factor of 2^{28}.

This document outlines the project background, design process, and performance/cost evaluation.

Theoretical Background

CCM is a means for efficiently evaluating the *log*, *exp*, and *square root* using simple operations [2, p. 572]. 'Shift' and 'add' are the main functions in using CCM. This cheap and iterative method can be implemented to facilitate the calculations of *log*, *exp*, and *square root* where their calculation is not the focus [1].

Design Process

The design process was broken into 3 phases: determining the execution requirements, analyzing the pseudocode, and implementing the algorithms. This section discusses each phase and the decisions made.

Design Requirements

In order to evaluate the performance and precision, design requirements were established. Below are the 4 main requirements.

REQ1 - The program must execute on a 32-bit ARM processor.
REQ2 - Calculations are limited to integer values.
    REQ2.1 - A scale factor of 2^{28} is used on integer values to facilitate computation.
    REQ2.2 - In converting to integer, values are rounded up if their decimal point value is greater than 0.5.
REQ3 - Calculations are done with 32-bits of precision.

REQ4 - C code must be optimized to produce minimal assembly code.

Analyzing the Pseudocode

Pseudocode was provided by Professor Sima [1]. Figures 1 through 3 show the pseudocode for calculating the binary logarithm, base-2 exponential, and square root. Here, K is 32, as REQ3 states the calculations are done with 32-bits of precision.

Calculation of Binary Logarithm – Pseudocode

1: $\{\log_2 M$ with K bits of precision$\}$
2: **for** $i = 0$ to $K - 1$ **do**
3: $\quad \mathsf{LUT}(i) = \log_2(1 + 2^{-i})$ {calculate the table with $\log_2 A_i$}
4: **end for**
5: $f = 0$
6: **for** $i = 0$ to $K - 1$ **do**
7: $\quad \mu = M \cdot (1 + 2^{-i})$ {potential multiplication by A_i}
8: $\quad \phi = f - \mathsf{LUT}(i)$ {potential addition with $\log_2 A_i$}
9: $\quad$ **if** $\mu \leq 1.0$ **then**
10: $\quad\quad M = \mu$ {if product is less than 1 accept iteration,}
11: $\quad\quad f = \phi$ {otherwise reject it (do nothing)}
12: $\quad$ **end if**
13: **end for**
14: **return** f

Figure 1: Pseudocode for CCM Binary Logarithm [1]

Calculation of Base-2 Exponential – Pseudocode

1: $\{2^M$ with K bits of precision$\}$
2: **for** $i = 0$ to $K - 1$ **do**
3: $\quad \mathsf{LUT}(i) = \log_2(1 + 2^{-i})$ {calculate the table with B_i}
4: **end for**
5: $f = 1.0$
6: **for** $i = 0$ to $K - 1$ **do**
7: $\quad \mu = M - \mathsf{LUT}(i)$ {potential addition with B_i}
8: $\quad \phi = f \cdot (1 + 2^{-i})$ {potential multiplication by 2^{B_i}}
9: $\quad$ **if** $\mu \geq 0$ **then**
10: $\quad\quad M = \mu$ {if sum is greater than 0 accept iteration,}
11: $\quad\quad f = \phi$ {otherwise reject it (do nothing)}
12: $\quad$ **end if**
13: **end for**
14: **return** f

Figure 2: Pseudocode for CCM Base-2 Exponential [1]

Calculation of Square Root – Pseudocode

```
1: {√M with K bits of precision}
2: f = 1.0
3: f_sqrt = 1.0
4: for i = 0 to K − 1 do
5:    μ = f · (1 + 2^−i) · (1 + 2^−i) {potential multiplication by A_i^2}
6:    μ_sqrt = f_sqrt · (1 + 2^−i) {potential multiplication by A_i}
7:    if μ ≤ M then
8:       f = μ {if product is less than M accept iteration,}
9:       f_sqrt = μ_sqrt {otherwise reject it (do nothing)}
10:   end if
11: end for
12: return f_sqrt
```

Figure 3: Pseudocode for CCM Square Root [1]

From analysis, the pseudocode can be improved to reduce the number of required calculations per iteration by one. In calculating the binary logarithm, ϕ is only required if the 'if' condition is met; similarly, ϕ and μ_sqrt are only carried over to the next iteration if the 'if' condition in the base-2 exponential and square root calculation are met. Thus, the calculations of ϕ and μ_sqrt can be done within the 'if' statements to reduce calculations per iteration, and the variables removed entirely.

Multiplication by $(1 + 2^{-i})$ is an operation that occurs frequently within the pseudocode. Through expansion, it was determined that $f \times (1 + 2^{-i})$ is equivalent to $f + (f \times 2^{-i})$, or $f + (f >> i)$. This calculation reduced a potential 4-cycle multiplication to 2-cycles: add and shift.

Implementing the Algorithms

The look-up table required by the binary logarithm and base-2 exponential algorithms was calculated in advance, since the number of bits of precision were known in advance. This allowed these values to be inserted as integers, as they are all values less than or equal to one: the scale factor of 2^{28} gave most of these values integer representation. Notably, the final 3 values in the look-up table evaluated to less than one. This suggests that some precision will still be lost in future calculations.

The algorithms were original written in the program as functions, but due to complications, they were embedded in the main function. K was also increased from 16 to 32 due to significant error occuring in the calculation of the square root. Unfortunately, this error could not be minimized further with the 32-bit limitation. Figures 4 through 6 show the implemented algorithms.

```
//binary logarithm
u-=u;
f1-=f1;
for(i-=i; i < k; i++) {
    u = (m1 + (m1 >> i));
    if(u <= sf) {
        m1 = u;
        f1 -= lut[i];
    }
}
```

Figure 4: CCM Binary Logarithm

```
//exponential
u-=u;
f2 = sf;
for(i-=i; i < k; i++) {
    u = (int32_t)m2-lut[i];
    if(u>=0) {
        m2 = u;
        f2 = (f2 + (f2 >> i));
    }
}
```

Figure 5: CCM Base-2 Exponential

```
//square root
i-=i;
u = 1073741824;
f3 = sf;
int32_t f_sqrt = sf;
while(i < k){
    if(u<=(int32_t)m3) {
        f3 = u;
        f_sqrt = (f_sqrt + (f_sqrt >> i));
    }
    i++;
    u = (f3 + (f3 >> (1-i)) + (f3 >> (i+i)));
}
```

Figure 6: CCM Square Root

Note that the algorithms have specific ranges for integer inputs: the binary logarithm calculation accepts values between 0.5 and 1.0, the base-2 exponential calculation uses a value from 0 to 1.0, and the square root calculation requires a value between 1.0 and 4.0.

Performance/Cost Evaluation

A testbench for the program was constructed. The program was to determine the binary logarithm of 0.75, the base-2 exponential of 0.3, and the square root of 1.4; with the scale factor of 2^{28} applied, these values became 201326592, 80530636, and 375809638. The expected outputs are -111410780 (-0.4150375), 330482812 (1.2311444), and 317617115 (1.18321596) respectively. Execution returned -111410784, 330482806, and 426674748. Coupled with analysis, this shows that the 32-bits of precision are enough to calculate the binary logarithm and base-2 exponential, but are not enough to precisely determine the square root as the add and shift operations rapidly approach multiplication by 1.

The most costly portions of the program were determined to be the 3 K-bit loops for computing the binary logarithm, exponential, and square root. As CCM relies on information from the previous iteration [2, pp. 577-578], the loops could not operate on decrementation. The second calculation in each loop was inserted into the 'if' statement, as the calculation was only required if the 'if' condition was met - essentially making predicate operations.

Based on findings in the pseudocode analysis, multiplication by $(1 + 2^{-i})$ became $f + (f >> i)$, while multiplying by $(1 + 2^{-i})(1 + 2^{-i})$ became $f + (f >> (1-i)) + (f >> (i+i))$ based on the expansion of the equation. This reduced 4-cycle multiplications to 2-cycle add-and-shifts, and 8-cycles of multiplication to 6-cycles of adding and shifting. Performing the multiplication this way also prevented issues in multiplication due to the scale factor.

Software pipelining was applied to the square root algorithm as the initial value of μ is known to be 4. This allowed all other calculations of μ to be done using adds and right-shifts, as the case where i is 0 requires a left-shift.

It was noted that having all the algorithms in the main function reduced parallelism. The code could be further optimized for parallel processing by separating the algorithms into individual functions.

Conclusion

The software optimization techniques that were applied to the CCM algorithms resulted in clear assembly code. With 32-bits of precision, binary logarithm and base-2 exponential were calculated with good precision. Square root calculations suffered from precision loss, and were not able to produce precise calculations given the system and software requirements.

Throughout the calculations, a hardware or firmware optimization was not observed. As such, a function that would significantly improve the program's performance was not designed.

Bibliography

[1] M. Sima, Lesson 102: Convergence Computing Method.

[2] S. Chen, C. Li and Y. Hou, "Compatible CORDIC and CCM algorithms for small area realization," VLSI Signal Processing, VIII, Sakai, Japan, 1995, pp. 572-578. [Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=527528&isnumber=11533]