University of Victoria
Summer 2018 SENG 440 Report

Report of 'Cordic Optimization'

Zhilun Liu, UVic, Dept. of CS, V00822606,charleslzl@hotmail.com
Linfeng Xu, UVic, Dept. of CS, V00824093, james2012.shun@gmail.com

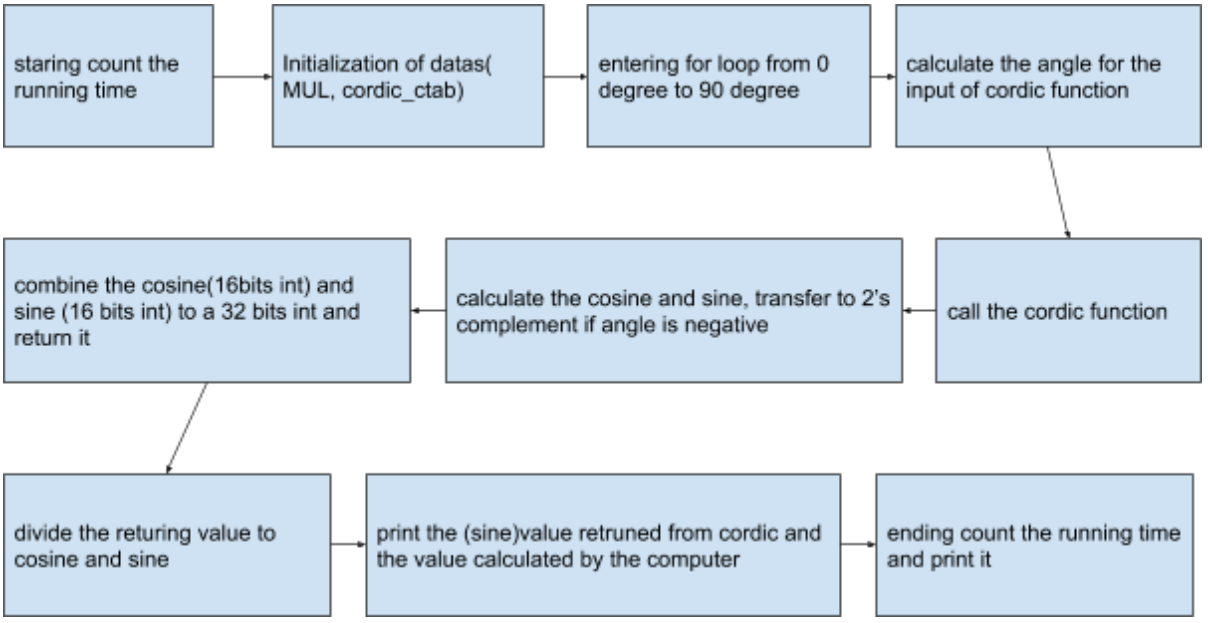
26 July, 2018

Introduction

Cordic is a simple and efficient algorithm computing the sine and cosine of a value using only basic arithmetic operation(addition, subtraction and shifts). In our project we will implement a simple CORDIC based on a source code and keep optimize it using different type of optimization techniques. We will also record the total instruction length and running time needed for different optimization techniques.

Design requirements and specifications (UML)

- Flow chart of the code:



Figuare 1 Activity flowchart for Pred_SP.c

- Design requirement of Cordic:
  - Reducing the running time of the code
  - The output of the cordic function should be nearly equal to the value that calculated by the computer

Theoretical Background

Cordic Algorithm

- CORDIC stands for “Coordinate Rotation Digital Computer” algorithm is a very efficient way of calculating the sine and cosine value of an angle using only addition, subtraction and bit shift instead of multiplication to calculate the rotation and orientation of a vector.

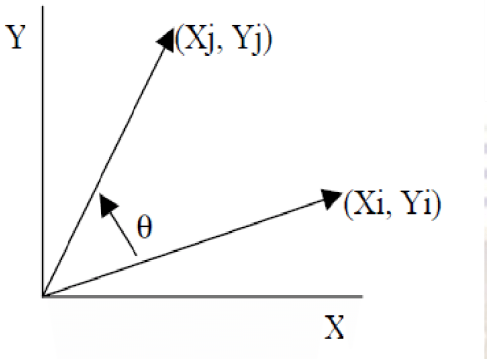


Figure 2 Cordic example

$$x_R = x_{in}cos(\theta) - y_{in}sin(\theta)$$
$$y_R = x_{in}sin(\theta) + y_{in}cos(\theta)$$

- If we choose Yi = 0 Xi = 1 after rotation we have Xj = cos(θ) Yj = sin(θ).
- The equation above can be transferred to

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = cos(\theta) \begin{bmatrix} 1 & -tan(\theta) \\ tan(\theta) & 1 \end{bmatrix} \begin{bmatrix} x_{in} \\ y_{in} \end{bmatrix}$$

- We need to perform 3 multiplications for every rotation. CORDIC provides two fundamental ideas to avoid the multiplication. The first one is that rotating the input vector by arbitrary angle θd that is equal to rotating the vector by several smaller angles, For example, 63.568 degree is same as three rotation by 50, 25.588 and -12.02 degree. However in this project we used the other method , we can write tan(θ) as 2^-i where i = 0,1,….,n. In this way we can achieve multiplication by bitwise shifting, which is much faster the multiplication.

Optimization techniques

- Operator strength reduction
  - The operators in high level language has significant difference in assembly code. Some operators feels like same in high level language, but it may take way more cycles than the other one. For instance, the operators, + (plus) ; * (times) ; / (divide). They take same space in C, however / (divide) will take 10 cycles, * (times) will take 3 cycles, and + (plus) only take 1 cycles. In order to optimize our code, to reduce programs running time, we would likely to replace high cost operators with lower cost one. This is the operator strength reduction.
- Loop unrolling

    - The loop unrolling is a space-time tradeoff technique that reduce the execution time by increasing code length. When we unfold the loop, the program will reduce the branch penalties which will take longer cycles, and the delay in reading data from memory.
- Software pipelining
    - Software pipelining is a loop optimization to make statements within an iteration independent of each other, and remove dependencies so that sequential instructions can execute in parallel. When a program does not using pipelining or the processor does not support pipelining, the instructions will fetch, decode, and execute linearly. When a pipelining program is running on a multi-thread processor, the instructions will fetch, decode, and execute parallely. Local variable
    - Using local variable can avoid many times of loading and storing datas from memory. However, we should be very careful about the number of the registers we used. Usually 8 to 9 registers is the maximum number of registers we can use.
- Pedicate operation
    - Eschewing branch predicate reduces the performance penalty brings by the wrong guess of processor. All the possible branches are executed in parallel before the branch condition is proved.
- Vector opreation
    - When we passing the data that generated by the cordic functions to the main functions, we combine them (2 16 bits integers) to one 32 bits integers, and divide them in the main function when we want to use the datas. This operation can save memory that needed.

Hardware Solution

Our project is designed and optimized for machine with ARM toolchain. In such hardware, this software can be execute in parallel and can reduce many cycles. However, it's not comes for free. There will be still at most 3 cycles latency.

The project requires the platform hardware is based on ARM processor, able to do pipelining execution, and support C language.

- If you want to access to a machine that have ARM toolchain in Uvic, you can login to campus network using "**ssh ugls.ece.uvic.ca**".ssh ugls.ece.uvic.ca
- How to compile and run in a ARM achitechture machine
  The platform needs "arm-linux-gcc" compiler to compile the project. To compile, user needs type "arm-linux-gcc -static -o filename.exe filename.c -lm". "-o" is a optimization flag, "-lm" is used to include math library.
- To run the output, type "**qemu-arm file.exe"**,then you will see the value calculated by our cordic function and the value calculated by the computer. After all you will see the total running time.

Firmware Solution

- For 1-issue slot solution, we have 559 lines assembly instructions in total

- For 2-issue slot solution, we have 438 lines assembly instructions in total(Opt_Pred_SP.s)
    - Because some instruction of the function is dependant which cannot be executed in parallel, therefore 2-issue slot solution only save about 23% cycles instead of 50% cycles
- For 3-issue slot solution we have 384 lines assembly instructions in total(3Opt_Pred_SP.s)
    - For same reasons, 3-issue slot solution only save about 32% cycles instead of 66% cycles.

Optimization datas

```
32 void cordic(int theta, int *s, int *c, unsigned long n){
33  double x = cordic_1K;
34  double y = 0;
35  double z = theta;
36  double v = 1.0;
37  unsigned long k;
38
39  for (k=0; k<n; ++k){
40          //int d = z >> 31;
41          int d = z>=0 ? +1 : -1;
42          double tx = x - d * v * y;
43          double ty = y + d * v * x;
44          double tz = z - d * cordic_ctab[k];
45          x = tx; y = ty; z = tz;
46          v *= 0.5;
47
48  }
49 printf("y is %f", y);
50  *c = x;
51  *s = y;
52 }
```

Figure3 cordic function of the very original code

```
 94 void cordic(int theta, int *s, int *c, int n)
 95 {
 96   int k, d, tx, ty, tz;
 97   int x=cordic_1K,y=0,z=theta;
 98   n = (n>CORDIC_NTAB) ? CORDIC_NTAB : n;
 99   for (k=0; k<n; ++k)
100   {
101     d = z>>31;
102     //get sign. for other architectures, you might want to use the more portable version
103     //d = z>=0 ? 0 : -1;
104     tx = x - (((y>>k) ^ d) - d);
105     ty = y + (((x>>k) ^ d) - d);
106     tz = z - ((cordic_ctab[k] ^ d) - d);
107     x = tx; y = ty; z = tz;
108   }
109  *c = x; *s = y;
110 }
```

Figure 4 cordic function of functinally correct C code

```
56 int cordic(int theta,int n)
57 {
58   register int k, d, tx, ty, tz,z_temp;
59   int32_t xy = 0;
60   int x=cordic_1K,y=0,z=theta;
61   int cordic_temp = cordic_ctab[0];
62   n = (n>CORDIC_NTAB) ? CORDIC_NTAB : n;
63   d = z>>15;
64   for (k=0; k<n; ++k)
65   {
66
67     //get sign. for other architectures, you might want to use the more portable version
68     //d = z>=0 ? 0 : -1;
69     z_temp = z - ((cordic_temp ^ d) - d);
70     cordic_temp = cordic_ctab[k+1];
71     tx = x - (((y>>k) ^ d) - d);
72     ty = y + (((x>>k) ^ d) - d);
73     tz = z_temp;
74     x = tx; y = ty; z = tz;
75     d = z>>15;
76
77
78     }
79
80     z_temp = z - ((cordic_temp ^ d) - d);
81     tx = x - (((y>>k) ^ d) - d);
82     ty = y + (((x>>k) ^ d) - d);
83     tz = z_temp;
84     x = tx; y = ty; z = tz;
85
86   xy = (xy | x)<<16;
87   xy = xy | y;
88   return xy;
89  // printf("x is %d, y is%d, xy is %d\n", x, y, xy);
90  //*c = x; *s = y;
91 }
```

Figure5 cordic function of the Pred_Sp.c

```
0.999939 : 1.000000
0.999939 : 1.000000
we need 90030000.000000 to complete.[charlesl@ugls44 Opt_methods]$
```

Figure6 output of Pred_Sp.c(Optimized code)

```
1.000000 : 1.000000
1.000000 : 1.000000
we need 109320000.000000 to complete.[charlesl@ugls44 p1]$
```

Figure7 output of cordic-test.c(original code)

	Instructions(lines)	Avg Running-time(gcc)	Avg Running-time(arm)
Functionally C code	69	13.36s	103.05s
Local_V	78	13.57s	93.27s
Loop_U	78	13.53s	92.63s
SP_Opt	78	13.49s	92.01s

Pred	40	12.55s	94.16s
Pred_SP	56	12.73s	91.50s

Q/A:

1. Why does SP takes more instruction than not using SP?
    a. In order to let processor running instructions parallelly in a fetch-execute cycle, program will has more instructions than no pipelining, but since it is running parallelly, it will take less cycles.
2. Why does Pred take more running time in arm compiler?
    a. when we compile using gcc compiler we set the optimization to O3, however, in ARM compiler we set to O.

Conclusion

From the observation of the datas, we can see that the Predicate operation with Software Pipelining is the best solution we got which saved about 10% running time. This because we used tried many techniques we learned from the SENG440, like software pipelining, Vector Operation, and using registers. Before doing this project, we have no background experience about the software optimization. However, we learned and implemented many techniques of the optimization through complete and design this project. In the cordic function, the goal is to get the 2's complement for every iterations. In this case, we can implement the predicate operation to avoid branches. We also used the Vector operation which combine the cosine and sine value into one 32 bits integer to return it to the main function. Overall, Cordic project is successful and is a excellent experience.

Functionally original code:

Simple C sourse code for CORDIC(2012), Retrived from http://www.dcs.gla.ac.uk/~jhw/cordic/index.html

Optimization code (C and Assembly):

https://github.com/ZhilunLiu/SENG440

Reference

1. Simple C sourse code for CORDIC(2012), Retrived from http://www.dcs.gla.ac.uk/~jhw/cordic/index.html
2. Predication(branch predication),(March 2011), Retrieved from https://whatis.techtarget.com/definition/predication-branch-predication
3. Slezica(19 June,2017), Retrived from https://stackoverflow.com/questions/44622572/do-static-local-variables-in-c-functions-affect-execution-speed
4. What is software piplineing(2 Sep. 2006), Retrived from https://insidehpc.com/2006/09/what-is-software-pipelining/

5. Steven Arar(31 Mar, 2017) Retrived from https://www.allaboutcircuits.com/technical-articles/an-introduction-to-the-cordic-algorithm/