UCSD 237-C: Project 2 Cordic Design
Lakshya LNU (A59017906)
Atharva Kshirsagar (A59019697)
Oct 24, 2023

Introduction

In this project, we are going to investigate various designs for CORDIC filter for the purpose of converting cartesian coordinates into polar coordinates. Specifically, we implement a baseline design using floating point numbers and then investigate effects of various design choices such as number of iterations, Data types, etc. on the accuracy, resources, latency and other metrics.

We also investigate a hash table(LUT) based design of the CORDIC, and compare that with the iterative CORDIC approach and show that LUT based CORDIC lacks in accuracy against the iterative approach.

CORDIC BASELINE

Our baseline design is based on floating pointing values:

```
#include "cordiccart2pol.h"

data_t Kvalues[NO_ITER] = {...};      // Values omitted for space
reasons

data_t angles[NO_ITER] = {...}; // Values omitted for space reasons


void cordiccart2pol(data_t x, data_t y, data_t * r,  data_t * theta)
{
#pragma HLS INTERFACE mode=s_axilite port=return
      // Write your code here
      int num_iter = NO_ITER;
      float curx = x;
      float cury = y;
      float cura = 0;

      // Rotate by 90
      int sigma = (cury > 0)? 1:-1;
      float tempx = curx;
      cura = cura + sigma*(1.5708);
```

```
        curx = sigma*cury;
        cury = -sigma*tempx;


        for(int i=0; i < num_iter; i++){
                sigma = (cury > 0)? 1:-1;
                tempx = curx;

                // Perform rotation
                curx = curx + cury*sigma*Kvalues[i];
                cury = cury - tempx*sigma*Kvalues[i];

                // update angle
                cura = cura + sigma*angles[i];
        }

        *theta = cura;
        *r = curx*0.60727;
}
```

Giving our baseline numbers of

Latency	Throughput(MHz)	BRAM	DSP	FF	LUT	r-RMSE	theta-RMSE
277	0.361	0	24	2849	4160	2.479e-5	1.975e-5

Q1: Number of Rotations

#Rotations	Latency	Throughput(MHz)	BRAM	DSP	FF	LUT	r-RMSE	theta-RMSE
11	202	0.495	0	24	2848	4159	2.453e-5	5.994e-4
12	217	0.461	0	24	2848	4159	2.474e-5	2.737e-4
13	232	0.431	0	24	2848	4159	2.479e-	1.377e-

							5	4
14	248	0.403	0	24	2848	4159	2.479e-5	8.231e-5
15	262	0.382	0	24	2848	4159	2.479e-5	4.217e-5

As we can see with increased number of rotations, the latency of the method increases, while the RMSE for theta reduces dramatically. We observed that RMSE for r loosely stays the same, after iteration 12, even though we increased the number of rotations. This happens if the number of significant digits in the output test bench are less than the number of precision bits used to represent those amount of significant digits. Thus, after a certain number of precision bits we stop seeing improvements. Also, as expected the resource utilization stays the same.

Q2: Data Type - cordic_optimized1

From the baseline code, we observed that we did not need to represent all variables in floating points. As the ultimate goal is to eliminate the multiplication operators in our baseline code, the first step is to convert floating points into fixed point types. Below is our code that does so -

```
#include "cordiccart2pol.h"

typedef ap_fixed<16, 2, AP_RND, AP_WRAP, 1> fixed_data_t;
typedef ap_fixed<16, 3, AP_RND, AP_WRAP, 1> cura_t;

fixed_data_t Kvalues[NO_ITER] = {...};

cura_t angles[NO_ITER] = {...};

void cordiccart2pol(data_t x, data_t y, data_t * r,  data_t * theta)
{
#pragma HLS INTERFACE mode=s_axilite port=return

      // Write your code here
      ap_uint<5> num_iter = NO_ITER;
      fixed_data_t curx = x;
      fixed_data_t cury = y;
      cura_t cura = 0;

      // Rotate by 90
      ap_int<2> sigma = (cury > 0)? 1:-1;
```

```
        fixed_data_t tempx = curx;
        cura = cura + sigma*((cura_t)(1.5708));
        curx = sigma*cury;
        cury = -sigma*tempx;


        for(int i=0; i < num_iter; i++){
              sigma = (cury > 0)? 1:-1;
              tempx = curx;

              // Perform rotation
              curx = curx + cury*sigma*Kvalues[i];
              cury = cury - tempx*sigma*Kvalues[i];

              // update angle
              cura = cura + sigma*angles[i];
        }

        *theta = cura;
        *r = ((data_t)curx)*0.60727;
}
```

Here, we convert our current x and y's to a 16-bit ap_fixed representation, and our current angles are also converted to a 16-bit ap_fixed representation as shown above. Correspondingly we change the types of our KValues and angles arrays.

Since different variables have different ranges, the number of bits and amount of floating point bits they require to be saved in the ap_fixed representations vary. Hence, for optimization purposes, we believe it is better for different variables to have different data types, depending on the ranges of those particular variables. Hence, the best data type, when defined customly as we have done above, does depend on the input values getting stored in these variables.

Our approach in converting data was to check the ranges of `x, y` and `cura`, and set the datatype to ones that satisfied the base minimum bit and number of integer and decimal bit requirements for each of these variables. Initially that was 2 integer bits total and 6 decimal bits for `x` and `y`, since the range of `x` and `y` is from 1 to -1. 3 integer bits and 6 decimal bits were used for `cura`. Decimal bits were chosen at random. Our RMSE in that case was quite poor as shown in the table below. So we changed our representation to 2 integer bits and 14 decimal bits for `x` and `y`, and 3 integer bits and 13 decimal bits for `cura`. We feel this is the best process for selecting data type -

1. Check ranges of variables required to be stored, calculate minimum integer and decimal bits required to store those.
2. Create fixed point types accordingly and check accuracy on these integer values.

3. If too low, more precision is required, hence increasing the number of decimal bits required accordingly, until acceptable performance is achieved.

Below is the results table we got after doing synthesis on the design described in the code above -

Latency	Throughput (MHz)	BRAM	DSP	FF	LUT	r-RMSE	theta-RMSE
105	0.952	0	13	2117	5907	4.613e-5	1.031e-4

As we can see, as compared to the baseline given in the introduction section, our latency has gone down drastically, as well DSP and FF usage has gone down. We are using more LUTs which we speculate are required due to us typecasting. RMSEs are not as good as baseline, but it is an acceptable tradeoff as our latency and resources used have reduced.

Q3: Add and shift - cordic_optimized2

We converted sigma variable from the previous question into if-else condition and replaced multiplication with the shift operation giving us following design

```
void cordiccart2pol(data_t x, data_t y, data_t * r,  data_t * theta)
{
#pragma HLS INTERFACE mode=s_axilite port=return

    // Write your code here
    ap_uint<5> num_iter = NO_ITER;
    fixed_data_t curx = x;
    fixed_data_t cury = y;
    cura_t cura = 0;
    fixed_data_t tempx = curx;

    // Rotate by 90
    if (cury > 0){
        cura = cura + ((cura_t)(1.5708));
        curx = cury;
        cury = -tempx;
    }else{
        cura = cura - ((cura_t)(1.5708));
        curx = -cury;
        cury = tempx;
    }
```

```
    for(int i=0; i < num_iter; i++){
        tempx = curx;

        if (cury > 0){
            // Perform rotation
            curx = curx + (cury >> i);
            cury = cury - (tempx >> i);

            // update angle
            cura = cura + angles[i];
        }else{
            // Perform rotation
            curx = curx - (cury >> i);
            cury = cury + (tempx >> i);

            // update angle
            cura = cura - angles[i];
        }
    }

    *theta = cura;
    *r = ((data_t)curx)*0.60727;
}
```

After this we were able to see following numbers

Latency	Throughput(MHz)	BRAM	DSP	FF	LUT	r-RMSE	theta-RMSE
56	1.78	0	11	2057	5942	0.0002080 7	0.0001136

We can see that the latency has reduced from 105 to 56 for the same clock period between cordic_optimized1 design in Q2 to this design. This showcases the benefits of using shift operation instead of multiplications in the design. As well as number of FFs and DSP reduced, however, number of LUTs increased slightly.
Notably the RMSE values have increased, we speculate that this due to in previous design Synthesis automatically using higher precision for storing intermediate values during calculations. i.e. in the code line `` `curx = curx + cury*sigma*Kvalues[i]` `` intermediate result `` `cury*sigma*Kvalues[i]` `` is stored in higher precision as compared to our case wherein, we use the same precision as the cury type.

CORDIC LUT

We are given a functionally correct CORDIC LUT implementation, which we have verified. Upon synthesis of baseline, that uses total 8 bit, with 2 integer bits for the inputs, we are able to see the following numbers

(Total, Int) Bits	Latency	Throughput(MHz)	BRAM	DSP	FF	LUT	r-RMSE	theta-RMSE
(8,2)	20	5	8	0	3	102	0.02309	0.05104

Question-4

Data-types:

The memory usage of the LUT is given by $S = P.2^{(M+N)}$ bits, wherein M, N are the number of bits in the x, y inputs, and P is the number of bits in the output.

Therefore, length and memory increases exponentially with the number of bits of input. While due to output the length of the LUT doesn't increase, but memory increases multiplicatively

Num of Bits:

- Since, x,y are b/w [-1,1] we require 2 bits to accommodate tcordiche range of 3.
- For theta the range is [-3.14, 3.14] therefore requiring 3 bits to cover range of 7
- For R, the range is [0,$\sqrt{2}$], therefore
    - If we use signed, then we need 2 bits
    - If we use unsigned, we can use only 1 bit

Accuracy wrt number of bits:

We experimented with the number of input bits to validate that higher the number of bits the lower RMSE value we can realize. Table below shows it

(W = Total, I = Int) bits for the input	RMSE(r)	RMSE(theta)
(7,2)	0.045045	0.09458

(8,2)	0.02309	0.05104
(9,2)	0.01187	0.02546
(10,2)	0.00605	0.01177
(11,2)	0.00306	0.00591

Performance:

We expected the performance of the model to slightly degrade if we increase the number of bits, due to additional operation. However, possibly due to operation chaining, we observe no effect of number of bits of input and output data types on the performance

(W = Total, I = Int) bits for the input	Latency(10 ns clock)
(8,2)	20
(9,2)	20
(10,2)	20

Comparison with the CORDIC_baseline:

Pros: Latency is the biggest pro for CORDIC LUT, as it requires no iterations.

	CORDIC_Baseline	CORDIC LUT
Latency	277	20

Cons:

- It cannot achieve the same accuracy as the CORDIC_baseline. The max number of bits with which we could run simulation on our 16 GB RAM laptop was just 12 total bits with 2 integer bits for the input. Anything above would give segmentation fault due to possibly high memory requirements. At 12 total bits for the input.
    - We were still seeing high RMSE values compared to CORDIC baseline as shown below.

	RMSE(r)	RMSE(theta)
CORDIC_LUT	0.001544	0.00295
CORDIC_baseline	0.00002479	0.00001975

- Simlarly, a single verdict to tell which design takes more memory, but such high number of BRAMs(table below) used in CORDIC_LUT indicates that it might be taking higher memory.

	BRAM	DSP	FF	LUT
CORDIC_LUT	2048	0	3	118
CORDIC_baseline	0	24	2597	3864

Overall, I would you CORDIC_baseline instead of CORDIC_LUT due to huge benefits in RMSE values.