CSE237C Project 2 - CORDIC

Henrik Larsson Hestnes

October 2021

1 CORDIC - Introduction

1.1 Baseline implementation

RMSE(R)	RMSE(Theta)	Throughput	BRAM	DSP	FF	LUT	URAM
0.000154387103976	0.000016765718101	861 KHz	0	21	1647	2708	0

This is the performance and resource usage of the baseline implementation. This baseline implementation will be used in the coming sections to compare different optimizations.
For this and the coming implementations, the test-bench will be slightly changed. The most important change is that the number of tests are doubled, to make sure to catch even more cases and edge cases. In addition to this, the golden theta output on the third test case was rotated by 2π. The reason for this is that the test-bench assumes that vectors in Quadrant III will be rotated counter-clockwise onto the positive x-axis. However, this implementation rather rotates these vectors clockwise onto the positive x-axis, to avoid nested if statements, and thus improve performance. In the polar coordinate system these angles are equivalent, so this is really only a matter of design choice. The same applies to the rest of the implementations.

1.2 Optimization 1 - fixed-point data types

RMSE(R)	RMSE(Theta)	Throughput	BRAM	DSP	FF	LUT	URAM
0.000393795664422	0.000562397006433	1.75 MHz	0	3	1005	3515	0

Compared to the baseline implementation, this optimization is obviously way better in terms of performance and resource usage. It more than doubles the throughput, uses 86% fewer DSPs, 39% fewer FFs and 30% more LUTs. The accuracy does however also drop quite drastically. In total, as long as the finest accuracy is not required, this can be looked on as an improvement and really emphasize the importance of doing operations on fixed-point data types instead of floats. This optimization still relies on some multiplication with the precomputed values in the arrays.

1.3 Optimization 2 - fixed-point and shift and add operations

RMSE(R)	RMSE(Theta)	Throughput	BRAM	DSP	FF	LUT	URAM
0.000138314135256	0.000416972296080	4.61 MHz	0	1	949	3516	0

In this implementation, all the internal data types in the function are fixed-point. In addition, every single multiplication with the precomputed values are changed with explixit writing of the shift and add operations, except the last multiplication to scale r right. As you can see, this drastically improves the throughput, and also improves the accuracy compared to Optimization 1.

2 Q&A

2.1 Question 1

One important design parameter is the number of rotations. Change that number to numbers between 10 and 20 and describe the trends. What happens to performance? Resource usage? Accuracy of the results? Why does the accuracy stop improving after some number of iterations? Can you precisely state when that occurs?

Rotations	RMSE(R)	RMSE(Theta)	Throughput	BRAM	DSP	FF	LUT
10	0.000156936934218	0.001306840451434	1.30 MHz	0	21	1646	2701
15	0.000154387103976	0.000029830362109	913 KHz	0	21	1646	2707
19	0.000154387103976	0.000002490995485	734 Khz	0	21	1647	2711
20	0.000154387103976	0.000000948325351	704 KHz	0	21	1647	2711

As can be seen from the table above, and in cordic_optimized3, the performance in terms of throughput naturally goes down with increasing amount of rotation, while the accuracy goes up. The resource usage does not change significantly with the different amount of rotations. Another interesting observation to make is that the accuracy of *theta* never seems to stop with more rotations(at least up to 20), while the accuracy of r saturates at 15 rotations or earlier. The reason for this is that the accuracy of r is directly dependent on the accuracy of x, which is directly dependent on the accuracy of y and $Kvalues$, whereas *theta* is only dependent on the accuracy of $angles$. Thus when the accuracy of the multiplication between x, y, and $Kvalues$ saturates, the accuracy of r will also saturate and more rotations do not help, while *theta* continues to converge towards the right solution. This is although not the case for every rotation, as one rotation may lead to a worse *theta*, but when you have enough test cases, more iterations will almost always lead to a lower RMSE on theta.

2.2 Question 2

Another important design parameter is the data type of the variables. Is one data type sufficient for every variable or is it better for each variable to have a different type? Does the best data type depend on the input data? What is the best technique for the designer to determine the data type(s)?

The data types of the variables is undoubtedly very important. The optimal solution would be to have one datatype tailor-made for every variable. However, it may be hard to achieve in reality, since it may be hard to know the exact range of every variable, and thus the best solution may be to have the same data type for variables that tend to be in the same range magnitude-wise. Variables that do not correlate or in any way has a connection with regards to magnitude should preferably have different data types, such as the increment variable in Optimization 1.

The best data type for a variable that is dependent on the input data obviously depend on the input data. However, that does not mean that it depends on the data type of the input data. The data type of the input data will often be decided by someone other than you, which you can do nothing about. It may therefore be smart to typecast the input data to some other data type before performing operations on it, as have been done in Optimization 1.

The best technique for the designer to determine the data types is to first have a look at what operations will be done to the variables of the data type, and then have a look at how the magnitude of the variables will range to get an idea of how big the data type has to be. After this the final "tuning" of the data type can be done incrementally through trial and error.

2.3 Question 3

What is the effect of using simple operations (add and shift) in the CORDIC as opposed to multiply and divide? How does the resource usage change? Performance? Accuracy?

Using simple operations such as shift and add instead of multiply and divide undoubtedly improves the performance, both in terms of throughput and resource usage, as Optimization 1 and 2 can confirm. By changing the variables from floats to fixed-point, the synthesizer changes the multiplication to shifts and adds, which can be confirmed by looking in the "Schedule Viewer", and leads to the improvement you can see in Optimization 1. By explicitly changing the multiplication in the code with right shift, which equals a division by two, we get an even improved throughput, and also a higher accuracy and lower resource usage. The only drawback with these optimization, which might be a critical one, is that the accuracy of the calculations naturally goes down when the bit-length of the variables goes down. However, in this case the decline in accuracy for this implementation is very close to negligible, so unless it is really desirable to have an extremely high accuracy CORDIC implementation, this optimization will be highly desirable.

2.4 Question 4

These questions all refer to the lookup table (LUT) implementation of the Cartesian to Polar transformation. How does the input data type affect the size of the LUT? How does the output data type affect the size of the LUT? Precisely describe the relationship between input/output data types and the number of bits required for the LUT. The way the input data size affects the size of the LUT is by its bitwidth i.e. resolution, and the range of magnitude the input data will be in. If you have a big range of magnitude you will need more entries in the LUT to achieve the desired accuracy, and if you have high resolution on the input data and want to keep the resolution through the conversion, you have to have enough entries in the LUT. The output data size affects the size of the LUT by the size of each entry. If you have high resolution, i.e. large bitwidth, every entry in the LUT should have the same bitwitdh, and the LUT therefore will become bigger. The relationship between the input/output data types is that the range of the magnitude of the input limits the range of the magnitude of the output in terms of r, which both will have a good impact on the size of the LUT. Also the resolution of the input directly limits the resolution of the output, both in terms of r and *theta*, and the size of these should be coherent, which will impact the size of the LUT.

The testbench assumes that the inputs x, y are normalized between [-1,1]. What is the minimum number of integer bits required for x and y? What is the minimal number of integer bits for the output data type R and Theta? When the inputs range from [-1, 1], the integer part can be 3 different digits; 1, 0 and -1. To represent these with two's compliment we need 2 integer bits. The greatest magnitude you can get from this is $\sqrt{2}$. Since the magnitude r always is positive, i.e. unsigned, and $\sqrt{2} \approx 1.41$, the integer part can only be 1 or 0, and can therefore be represented by 1 integer bit. For theta, as long as you always uses the shortest rotation part to the positive x-axis, the greatest theta value is $\pm\pi \approx \pm3.14$. The integer part can therefore be 3, 2, 1, 0, -1, -2 or -3, which has to be represented by 3 integer bits. The number of integer bits required to represent theta is however independent of the magnitude of the input data. With 3 integer bits for theta you can always represent every point in the polar coordinate system.

Modify the number of fractional bits for the input and output data types. How does the precision of the input and output data types affect the accuracy (RMSE) results? What is the performance (throughput, latency) of the LUT implementation. How does this change as the input and output data types change?

Fractional bits	RMSE(R)	RMSE(Theta)	Throughput	BRAM	DSP	FF	LUT
3	0.038790836930275	0.741694271564484	76.8 MHz	2	0	13	152
6	0.023094084113836	0.051045715808868	76.8 MHz	8	0	3	104
9	0.023094084113836	0.051045715808868	76.8 MHz	512	0	3	90

As expected, the fewer fractional bits i.e. resolution, the less accurate results. This effect does however saturate some place between 3 and 6 fractional bits. After this, the only effect of more fractional bits is more resource usage, without higher accuracy, thus simply a waste. The throughput however does not seem to be impacted by the fractional bits.

What advantages/disadvantages of the CORDIC implementation compared to the LUT-based implementation? One big advantage of the CORDIC implementation is the accuracy, as can be seen by comparing the LUT table with the other above. However, the throughput and resource usage of the LUT implementation is far better than what was achieved by the CORDIC implementation.

3 PYNQ Demo

The testbench in the .ipynb had to be slightly changed. The reason for this is, as I stated in the baseline implementation, that the test bench assumes that points in Quadrant III will have a negative theta, while my solution gives them the positive theta value which is rotated 2π, and is really only a matter of definitions. The two points are totally equivalent, and I therefore took the liberty to change the test bench.