Optimizing CORDIC on an ARM Processor

Noah Clarke - V00883425 - noahwclarke@uvic.ca
Andrew Tran - V00820622 - trandrew@uvic.ca

August 14, 2020

Table of Contents

1. Introduction

The goal of this project is to implement the CORDIC Vectoring Mode algorithm on an ARM processor, and to optimize the implementation to minimize the number of assembly level operations per iteration.

The Background section will present the theory behind CORDIC. In the design section we will describe the various ways we optimized the implementation. The discussion section will review and analyze how well our design worked. Finally, the conclusions and future work sections will summarize the results of the project and present ideas for improving the project in the future.

1.1 Contributions

Noah Clarke

- High level C code optimization
- Research into ARM NEON
- Report writing/editing
- Final implementation of pipelined CORDIC algorithm using ARM NEON intrinsics
- Investigation into hardware assist for CORDIC
- Creation of diagrams

Andrew Tran

- High level C code optimization
- Research into ARM NEON
- Initial implementation of CORDIC using ARM NEON intrinsics
- Benchmarking of various implementations of CORDIC
- Creation of diagrams

1.2 Building Command and Platform Information

Command used to compile executable on ARM VM:

```
gcc -O3 cordic_main.c cordic_V_fixed_point.c -o cordic.out -lm
```

Note the `mfpu` flag is required when compiling a version of the CORDIC code which uses ARM NEON instructions:

```
gcc -O3 -mfpu=neon -g cordic_main.c cordic_V_fixed_point_with_neon.c -o cordic.out
-lm
```

Command used to generate assembly code on ARM VM:

```
gcc -O3 -S -mfpu=neon cordic_V_fixed_point.c
```

All code written, tested and benchmarked for this report was compiled and executed on the ARM virtual machine provided for the SENG 440 course. The VM has the following specifications:

```
OS: Fedora 29
Linux Kernel: 4.18.16-300
Platform: ARMv7 HL
```

For consistency, all benchmarks were completed with the VM running on the same host machine.

2. Background

CORDIC is an algorithm which rotates vectors from one starting angle to another target angle. More specifically, CORDIC aims to be more performant than the typical method of using trigonometric functions to solve for the rotation of a vector [1]. Using CORDIC one can avoid computationally demanding trigonometric functions and floating point numbers. The CORDIC algorithm has two modes, vectoring mode and rotation mode [1]:

Rotation mode: A vector is input as well as an angle to rotate the vector by. The new components of the vector after the rotation are returned [2].
Vector mode: A vector is input. The vector is subsequently rotated until it is aligned to the y-axis (i.e. the Y component of the vector is zero). The X component (which is also the magnitude of the vector) and the angle the vector was rotated by are returned [2].

For this report we will only focus on the implementation and optimization of vectoring mode. Thus, with any mention of the CORDIC algorithm in this report, we are specifically referring to the vectoring mode. It should be noted that converting between a vectoring mode and rotation mode implementation only requires trivial changes.

2.1 CORDIC Vectoring Mode

The vectoring mode algorithm takes two arguments, an X-value and a Y-value, and it outputs three values: a new X (which is also the magnitude), a new Y (which is zero while cordic is in vectoring mode), and the angle Z that the input vector (X, Y) initially made with the x-axis. In other words, the goal of the algorithm is to efficiently calculate arctan(Y/X) by only using cheaper operations such as shift, add and subtract [2].

For improved performance, the algorithm makes use of a pre-determined look-up table of elementary angles (one entry per iteration, see Figure 1). The algorithm works in a feedback loop, rotating the input vector (X, Y) by each elementary angle in the look-up table, either clockwise or counter-clockwise depending on if the current Y-value is positive or negative with the goal of reducing Y to zero [1]. The value Z also increments or decrements by each elementary angle to track the current rotation applied to the input vector [1]. After iterating *n* times (n = # of table entries), the final (X, Y) vector should ideally be in-line with the x-axis, and the Z value that was incremented/decremented *n* times will contain an approximation of arctan(Y/X). See Figure 2 for a flow diagram of the CORDIC algorithm.

Iteration *i*	Elementary angle degrees
0	45.00
1	26.56
2	14.04
3	7.13
4	3.58
5	1.79
6	0.89
7	0.45

Figure 1. Example Look-up Table with 7 entries

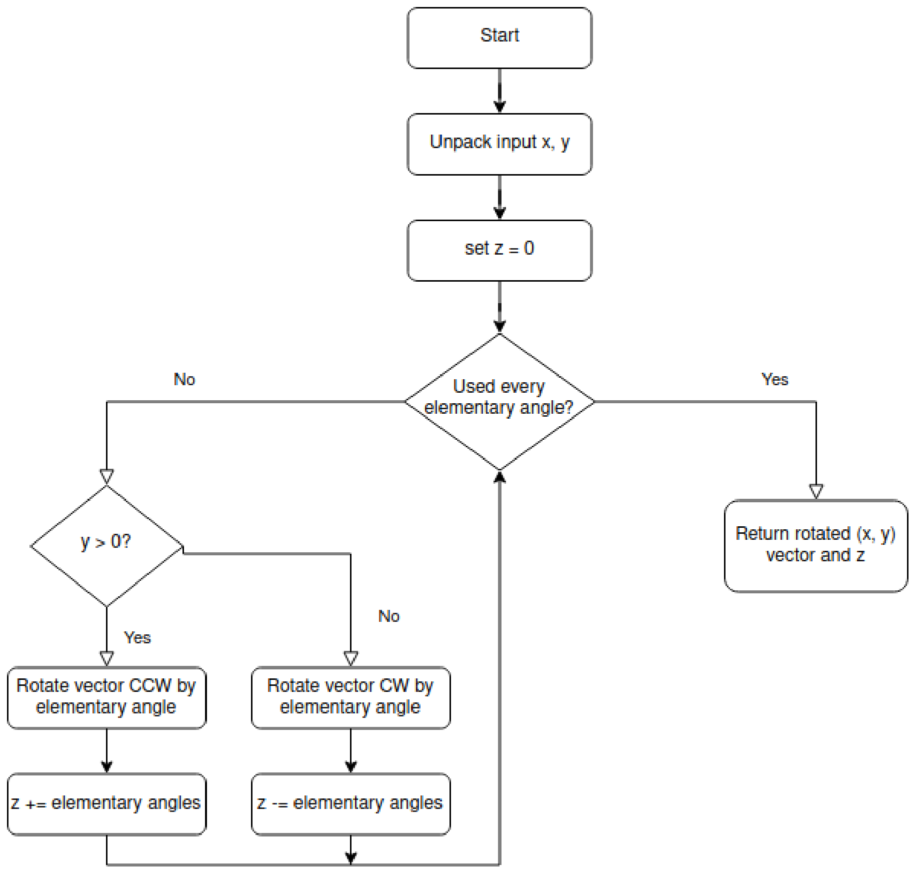

Start
Unpack input x, y
set z = 0
Used every elementary angle?
No
Yes
Return rotated (x, y) vector and z
y > 0?
Yes
No
Rotate vector CCW by elementary angle
Rotate vector CW by elementary angle
z += elementary angles
z -= elementary angles

Figure 2. Flow Diagram of CORDIC Algorithm

3. Design

The primary goal of this project was to investigate potential optimizations to an implementation of the CORDIC algorithm in C. First, we took the demo C code implementation of CORDIC from the SENG440 lecture slides. The code worked as expected, and correctly calculated arctan(y/x) for the example input values. An inspection of the assembly code generated by the reference implementation yielded three observations of note:

1. Each iteration of the for-loop only required 10-11 instructions
2. While using the -O3 flag the compiler performed loop unrolling on the entire application
3. No costly branch or jump instructions were present

Points 2 and 3 were of special interest as they demonstrate reference implementation is already somewhat optimized. Throughout our optimisation we took care to keep those points in mind to avoid any performance regressions. This provided a base reference implementation to work from. Next we edited the code in various ways, attempting to optimize the performance of the function by implementing various optimization techniques, including:

- Replacing the if-else statement with different logic
- Manually Unrolling the loop
- Using Software Pipelining
- SIMD Operations to calculate two values in parallel
- Allocating registers for frequently used variables
- Designing a hardware assist which provides a new machine instruction

Given that the majority of the CORDIC's operations happen within a single for-loop, loop optimizations were our primary target.

3.1 Optimization Process

The principal focus of this project was to improve the runtime of the CORDIC algorithm, therefore, it was essential to track how changes to the code affected performance. After each change manual assembly code inspection and/or benchmarks were completed to track our progress in optimizing the algorithms implementation.

The short length of the CORDIC algorithm's implementation (~35 LOC in C) made manual inspection of the assembly code each change easily possible. Additionally, we also used the Linux utility perf to track the runtime of the algorithm. While not precise this crude timing method was useful in determining if changes made to the code had in general a positive or negative impact on the runtime.

3.2 Packing and Unpacking of Input and Return Values

Our first modification to the CORDIC code was to implement variable packing and unpacking to facilitate the potential creation of an ARM subroutine in the future. The CORDIC algorithm implementation we are using requires three 16-bit values, x, y, and z, as input which are then modified by the algorithm and then returned as X', Y', and Z'. ARM platform subroutines, however, support receiving two 32-bit inputs and returning one 32 bit value [3]. To circumvent this limitation we packed the 16-bit values X and Y into a single 32 bit variable before passing that single value to the CORDIC subroutine. In the CORDIC subroutine the 32-bit value is unpacked. When the new values of X and Y are to be returned from the subroutine they are again packed into a single value which is returned and unpacked by the calling function. Bit shift and logical OR were used to pack and unpack the values. The third value required by CORDIC, z, was passed by reference. This solves the problem of needing to return 48 bits of data (3 values at 16 bits each) while only having 32 bits to work with. By passing by reference we also avoid the extra work of packing/unpacking one variable.

```
x_temp_1 = xy & 0xffff;
y_temp_1 = (xy >> 16) & 0xffff;
```

Unpacking of the X and Y components

```
return y_temp_1 << 16 | x_temp_1;
```

Packing of the X and Y components

Compared to the original implementation the new version is expected to be marginally slower due to the additional overhead from unpacking input and packing return values. The slowdown, however, is insignificant given it is only two bit shift operations and three logical AND/OR operations.

3.3 Branch Reduction

The branching which occurs in the for-loop of the CORDIC program was our first target for optimization given how obvious it is. Originally the majority of the code in the for-loop is divided into two branches as follows:

```
for( i=0; i<15; i++) {
      if( y_temp_1 > 0) {
        x_temp_2 = x_temp_1 + (y_temp_1 >> i);
        y_temp_2 = y_temp_1 - (x_temp_1 >> i);
        z_temp += z_table[i];
      } else {
        x_temp_2 = x_temp_1 - (y_temp_1 >> i);
        y_temp_2 = y_temp_1 + (x_temp_1 >> i);
        z_temp -= z_table[i];
      }
      x_temp_1 = x_temp_2;
      y_temp_1 = y_temp_2;
    }
```

It is immediately obvious this branching is less than ideal and an obvious candidate for improvement. The code in each branch is actually almost identical, from one branch to another only the operators are swapped (addition is swapped for subtraction and vice versa). Given this insight the branched code could be collapsed into a single path with a coefficient of 1 or -1 on the second operand being used to simulate the swapping of addition and subtraction operation. A ternary operator was used to determine the coefficient.

```
int sign;

    for( i=0; i<15; i++) {
```

```
    sign = y_temp_1 > 0 ? 1 : -1;

    x_temp_2 = x_temp_1 + (sign*(y_temp_1 >> i));
    y_temp_2 = y_temp_1 - (sign*(x_temp_1 >> i));
    z_temp += sign*z_table[i];

    x_temp_1 = x_temp_2;
    y_temp_1 = y_temp_2;
  }
```

Initially it may appear that the new version using the ternary operator may be slower due to the additional multiplication operations. This change, however, was completed with the knowledge that ARM includes multiply-accumulate (MLA) and multiply-substract operations (MLS) meaning the change therefore should have an insignificant impact on the number of instructions per loop iteration [4].

While this change was done with the goal of improved performance by potentially reducing branching (this ended up not being the case, refer to section 4.2 for more details). A secondary motivation was that the removal of the if-else construct might make future optimizations much easier. The introduction of the `sign` variable allowed the programmer some freedom to choose when the checking of the sign of the Y component takes place.

3.4 Manually Unrolling the For-Loop

Loop unrolling is an optimization technique where a while-loop or for-loop is altered, such that it performs duplicated operations per iteration. For example, if a loop updates an X variable each iteration, it could be unrolled such that X is updated twice per iteration, or 3 times per iteration, and so on. Unrolling a for-loop increases the code size but can also make the code run faster.

While the compiler automatically unrolls the for-loop when using the -O3 flag, the motivation for manually unrolling the loop was to hopefully incite the compiler to perform some additional optimizations. We implemented a version of the CORDIC function in which the loop is unrolled once per iteration, such that X, Y, and Z are updated twice during each iteration. We compiled the basic CORDIC code and the manually unrolled code using the -O3 flag. Refer to Appendix A for the C source code.

As expected, the assembly code was virtually unchanged compared to previous versions when compiling using the -O3 flag, therefore there was an equal amount of instructions per iteration. The manually unrolled version had an average execution time of 22.5 ms, whereas the ternary operator version it was based on took an average 22.7 ms, a statistically insignificant change. Additionally, the added length and repetition in the code made it rather unwieldy to update for future optimizations. Therefore, we chose not to pursue loop unrolling any further.

3.5 Software Pipelining

In a further attempt to optimize the for-loop, software pipelining was implemented on top of the branch reduction. With software pipelining the goal is to reorganize instructions within a loop (and potentially across iterations of the loop) in such a way that dependencies between instructions are minimized [5]. In theory, this should allow the processor to spend less time waiting for specific instructions to be computed. Please refer to Appendix A for the source code.

```
int cordic_V_fixed_point(int xy, int *z) {
  int x_temp_1, y_temp_1, z_temp;
  int x_temp_2, y_temp_2;
  int i=0;

  x_temp_1 = xy & 0xffff;
  y_temp_1 = (xy >> 16) & 0xffff;
  z_temp = 0;

  /* loop prologue */
  int sign = y_temp_1 > 0 ? 1 : -1;

  x_temp_2 = x_temp_1 + sign*(y_temp_1 >> i);
  y_temp_2 = y_temp_1 - sign*(x_temp_1 >> i);
  z_temp += sign*z_table[0];

  for( i=1; i<15; i++) {
       x_temp_1 = x_temp_2;
       y_temp_1 = y_temp_2;

       sign = y_temp_1 > 0 ? 1 : -1;

       x_temp_2 = x_temp_1 + (sign*(y_temp_1 >> i));
       y_temp_2 = y_temp_1 - (sign*(x_temp_1 >> i));
       z_temp += sign*z_table[i];
  }

  /* loop epilogue */
  x_temp_1 = x_temp_2;
  y_temp_1 = y_temp_2;

  *z = z_temp;

  return y_temp_1 << 16 | x_temp_1;
}
```

By introducing a loop prologue and epilogue we hoped to improve the performance of the first and last iterations of the loop by discarding unneeded instructions. Making improvements to the loop itself proved difficult, however, due to its short length and because the operations of the next loop iteration rely on the results of the last iteration.

3.6 SIMD Operations

SIMD operations allow for the optimization of many loops by effectively computing multiple iterations of the loop in parallel [6]. For this to be achieved, however, there must be no dependencies between loop iterations. Unfortunately, CORDIC does have dependencies between iterations so this typical optimization is not available to us. Instead we used SIMD operations to optimize the computation of a single loop iteration.

Given how the calculation of the new X and Y components of the vector are nearly identical, calculating them in parallel using SIMD operations appeared to be a promising candidate for improving performance. ARM provides NEON intrinsics for performing SIMD computations. NEON intrinsics act as normal ARM instructions but take in and return vector-like objects containing one, two, four, or eight values instead of a single value. The intrinsic applies the same operation to each value [7].

The following diagrams provide an overview of how X and Y are calculated for one iteration and how that process was parallelized using NEON intrinsics.

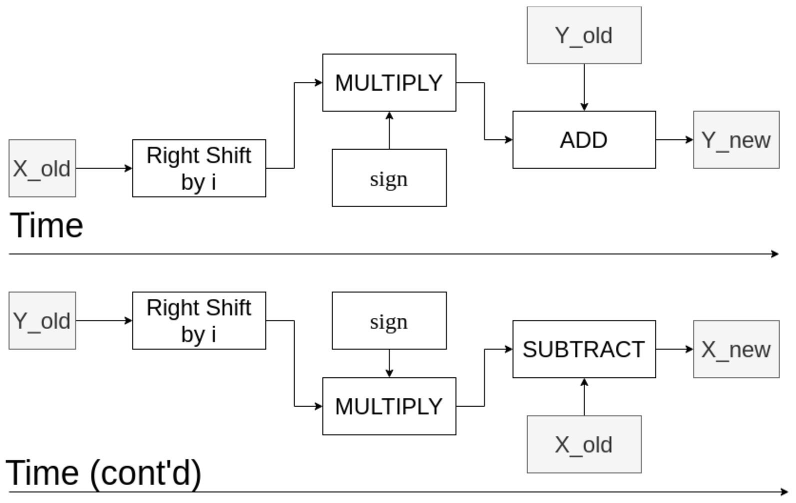


Figure 3: Computation of new X and Y component values. Note the computation of new X and Y values occurs sequentially.

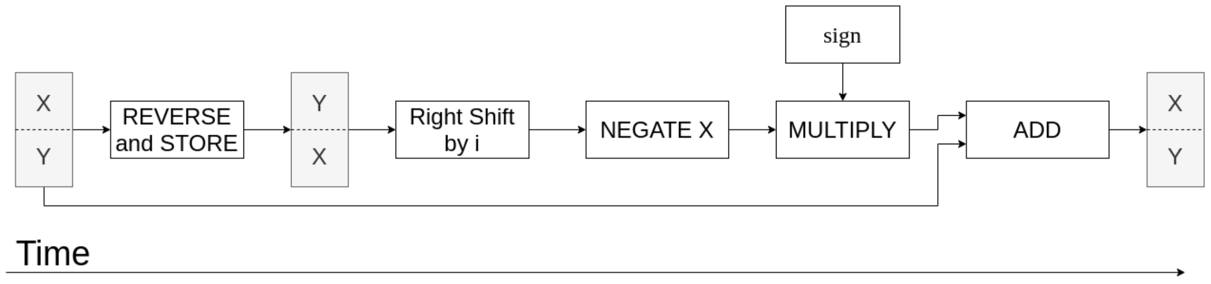


Figure 4: Computation of new X and Y component values using NEON intrinsics.

The code for the SIMD version of CORDIC is shown below:

```
int cordic_V_fixed_point(int xy, int *z) {
  int i = 1;
  int32_t sign = (xy >> 16) & 0xffff > 0 ? 1 : -1;

  int32_t z_temp = 0;

  int32x2_t xy_neon = {xy & 0xffff, (xy >> 16) & 0xffff};

  int32x2_t yx_neon = {(xy >> 16) & 0xffff, -(xy & 0xffff)};

  z_temp += sign*z_table[0]; // compute z

  for(; i<15; i++) { /* 15 iterations are needed */

    xy_neon = vmla_n_s32(xy_neon, yx_neon, sign); // add

    sign = xy_neon[1] > 0 ? 1 : -1; // get new sign

    yx_neon = vshr_n_s32(vrev64_s32(xy_neon), i); // copy and shift

    z_temp += sign*z_table[i]; // compute z

    yx_neon[1] = -yx_neon[1]; // make addition subtraction for y

  }

  xy_neon = vmla_n_s32(xy_neon, yx_neon, sign); // add

  *z = z_temp;

  return xy_neon[1] << 16 | xy_neon[0];
}
```

An explanation of the functionality of the used NEON intrinsics can be found in Appendix C.

While the usage of NEON intrinsics reduces the number of arithmetic operations required to compute the new values of X and Y it does unfortunately add a new operation:

```
yx_neon[1] = -yx_neon[1];
```

This is done so that when an addition is used to compute the new value of X a subtraction is used to calculate the new value of Y or vice versa.

3.7 Allocating Register for Commonly Used Values

Seeing the variable `i` is used frequently as both the loop counter and amount to bit shift right it should likely reside in a register file. Therefore we tried using the **register** specifier when declaring the variable `i`. After compiling the function using no optimization flags, it was observed that the compiler listened to our request and stored the loop counter in a register. This resulted in the assembly code containing 10 fewer operations (ldr and str) altogether, and therefore would reduce the latency by at least 10 cycles, when compiling the code using no optimization flags.

Next we compiled the code using the -O3 flag and the register specifier. As expected, the assembly code did not change, as the compiler optimized the code by unrolling the loop, such that a loop counter variable was no longer required. Thus there was not a change in performance when compiling the code using the -O3 flag.

3.8 Proposed Hardware Assist

To this point our focus has been on software level improvements to the CORDIC code. We will now investigate how a new hardware module (and thus assembly instruction) might provide a much more significant improvement to the performance of the CORDIC algorithm.

Initially we considered implementing the entire CORDIC algorithm in hardware. However, after spending a significant amount of time inspecting the assembly code for CORDIC we arrived at the conclusion that a hardware assist for computing new values of X and Y would be more appropriate. This alternative hardware assist would still provide a significant performance boost while having the benefit of being a much simpler and in theory cheaper to adopt.

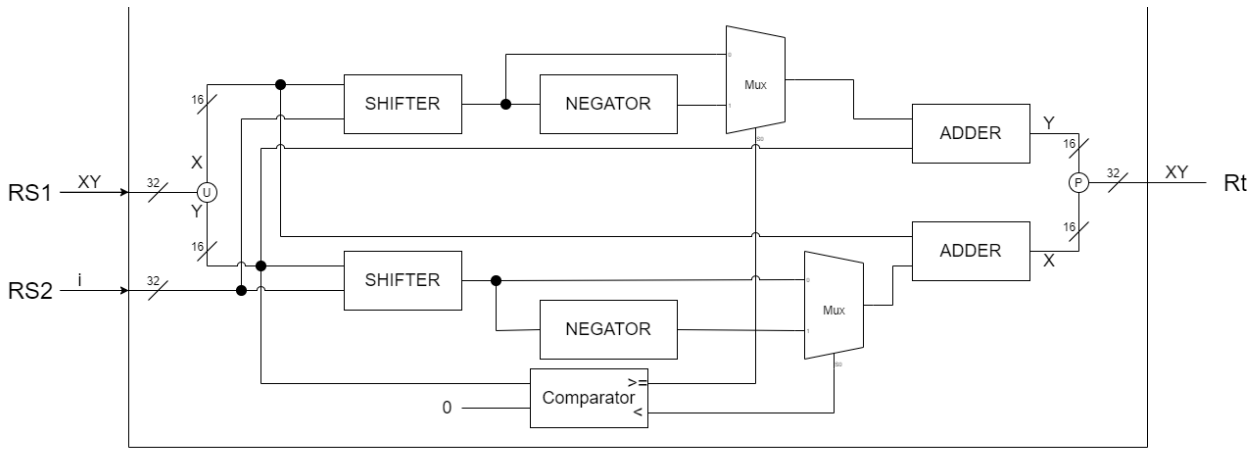


Figure 5: Diagram of proposed hardware assist for CORDIC.

The proposed hardware assist is relatively simple. The most complex component would likely be the s16-bit barrel shifters [8]. The encircled 'U' and 'P' represent wiring to facilitate the packing and unpacking of the X and Y components. The hardware assist takes X and Y components packed into a single register and i (the shift amount) as input. As output it produces a register packed with the new X and Y components.

```
#include <stdio.h>

int z_table[15];

int cordic_V_fixed_point(int xy, int *z) {
  int z_temp;
  int i;

  z_temp = 0;

  int sign;

  for( i=0; i<15; i++) { /* 15 iterations are needed */

    sign = xy > 0 ? 1 : -1;

    __asm__ __volatile__ (
      "cordic_hw\t%0, %1, %2\n"
      : "=r" (xy)
      : "r" (xy), "r" (i)
    );

    z_temp += sign*z_table[i];

  }

  *z = z_temp;

  return xy;
}
```

By allowing our new instruction to receive packed X and Y components we avoid any overhead related to packing or unpacking coordinates in software. Moreover, by packing the Y component into the most significant bits of XY we can compare XY directly to zero as Y and XY will have identical signed bits.

4. Discussions

The discussion will include a review and analysis of the work completed in the design phase. Additionally, discoveries made during the design phase will be further explored.

4.1 Comparing Execution Speed

We compared the execution time of each version of CORDIC using the perf tool in Linux. This tool allows us to measure the execution time of user-level instructions alone, so that there is almost no noise from OS level instructions. We created different executables from each version of CORDIC C code, using no optimization flags. For example, to compile the base version of CORDIC we use the following command:

```
gcc basic.c main.c -o basic.exe -lm
```

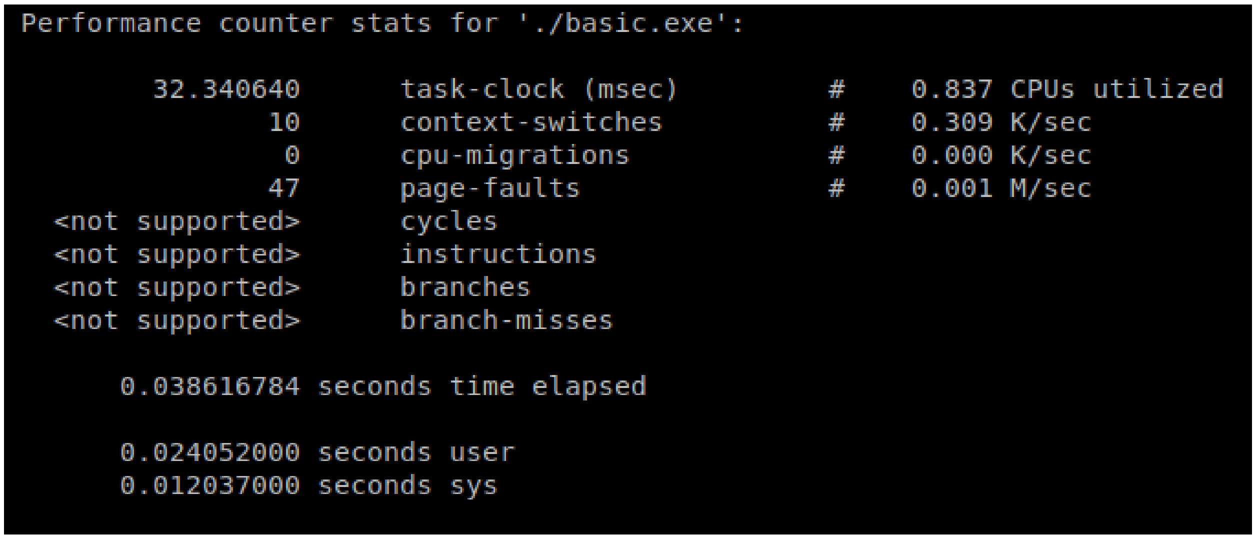
Next we used the perf tool on each executable 5 times, and took the average “seconds user” value returned from perf to get an average execution time (See Figure 3). For example, to get the execution time for the basic.exe version, we run the following command:

```
perf stat ./basic.exe
```

The average execution time for each version of CORDIC is shown below in Table 1. From the table it appears there are marginal improvements in execution time from changing the C code. It seems that implementing software pipelining, removing branches, and implementing SIMD instructions have a small impact on performance. For the manually unrolled code, the execution time is effectively unchanged. It’s important to note that this method for measuring performance is not perfect, because the execution time of each program was slightly random. However, the range of execution times of each program only varied by about 3 ms or so at most, so the execution times have some consistency.

CORDIC Version	Avg Execution Time	Execution Time Improvement over Basic version
basic.exe	22.7 ms	-
unroll.exe	22.5 ms	0.881%
pipe.exe	21.3 ms	6.17%
branchless.exe	21.2 ms	6.61%
neon.exe	21.5 ms	5.29%

Table 1. Average Execution Time for Each CORDIC Variation



```
Performance counter stats for './basic.exe':

       32.340640      task-clock (msec)         #    0.837 CPUs utilized
              10      context-switches          #    0.309 K/sec
               0      cpu-migrations            #    0.000 K/sec
              47      page-faults               #    0.001 M/sec
   <not supported>      cycles
   <not supported>      instructions
   <not supported>      branches
   <not supported>      branch-misses

     0.038616784 seconds time elapsed

     0.024052000 seconds user
     0.012037000 seconds sys
```

Figure 6: Perf Tool Output

4.2 Analysis of Software Optimizations

Overall only small gains in performance were achieved through software optimizations. Given the short length of the CORDIC algorithm and the already optimized reference implementation this is understandable. See section 4.3 for further analysis of the difficulties encountered.

4.2.1 Loop unrolling

As previously stated, the compiler automatically unrolls the for-loop for the reference implementation of CORDIC, when using the -O3 flag. Therefore, witnessing no real performance gain from unrolling the loop manually was expected.

4.2.2 Replacing if-else With the Ternary Operator

Replacing the if-else construct with the ternary operator gave a modest performance bump according to our timings (22.7ms to 21.2ms). Comparing the assembly code of the reference implementation to that using the ternary operator gives some insight and seems to confirm an improvement may have been made.

Original Implementation	Ternary Operator Implementation

`cmp    r3, #0`
`ldr    lr, [r2, #36]`
`subgt  r3, r3, r4`
`addle  r3, r3, r4`
`addgt  r0, r0, r5`
`suble  r0, r0, r5`
`asr    r4, r0, #10`
	`cmp    r0, #0`
	`movgt  r5, #1`
	`mvnle  r5, #0`
	`asr    r3, lr, #9`
	`mla    r3, r4, r3, ip`
	`asr    lr, r3, #10`
	`mls    lr, r5, lr, r0`

asr r5, r3, #10 addgt ip, ip, lr suble ip, ip, lr	mov r10, r5

Both implementations use the same number of instructions per loop iteration, note the additional conditional instructions in the reference implementation. Also note that as expected the multiply-accumulate (MLA) and multiply-subtract (MLS) instructions are used. The implementation using the ternary operator, however, eliminates the potentially costly LDR instruction which accesses memory [9]. Combined with the quality of life improvement of working with code without an if-else construct made this change a distinct upgrade to our CORDIC implementation.

4.2.3 Software Pipelining

Due to the structure of the CORDIC algorithm our ability to improve the existing code was limited. Pipelining simply involved adding a loop prologue and epilogue. These limited changes were reflected in the unchanged performance compared to the ternary operator version which was used as a starting point. This was confirmed by the effectively unchanged assembly code as well.

4.2.4 Parallelization of X and Y Calculation with SIMD

Disappointingly, implementing ARM NEON intrinsics to facilitate SIMD processing had a neutral to negative effect on performance. Comparing the assembly code for one loop iteration between the implementation using the ternary operator and that using NEON intrinsics provides some insight.

Ternary Operator Implementation	SIMD Implementation with NEON Intrinsics
cmp r0, #0 movgt r5, #1 mvnle r5, #0 asr r3, lr, #9 mla r3, r4, r3, ip asr lr, r3, #10 mls lr, r5, lr, r0 mov r10, r5	cmp r3, #0 vmov.32 r3, d17[1] movgt ip, #1 mvnle ip, #0 vmov.i32 d7, #0 @ v2si rsb r3, r3, #0 vmov.32 d7[0], ip vmov.32 d17[1], r3 vmla.i32 d16, d17, d7[0] vrev64.32 d17, d16 vmov.32 r3, d16[1] vshr.s32 d17, d17, #11

Unfortunately the overhead needed to facilitate the usage of NEON intrinsics, specifically the negation of the Y component, in addition to the less graceful memory management (notice the extra VMOV.i32 instructions) meant that NEON did not prove to be a viable way to improve performance. However, when reimplementing CORDIC with NEON intrinsics a less conventional optimization became apparent, see section 4.5. For this reason while attempting to optimize CORDIC with SIMD processing did not yield the result we initially hoped for it did provide insight which made the exercise worthwhile.

4.3 Difficulty Encountered with Software Optimizations

Our attempts to optimize CORDIC were been constrained by two key issues:

1. The CORDIC algorithm itself is quite short. The entire implementation can be written in ~35 lines of code. Moreover, the actual loop where the bulk of the computation takes place only executes ~10 operations. This small size limits the number of candidates for optimization.
2. “Looking ahead” in the CORDIC algorithm is not possible as the operations executed in one iteration depend on the results of the last iteration (in particular the sign of the Y component).

Due to these constraints we determined making large performance improvements with only high level C code changes is highly difficult or not possible. Thus we turned to other possibilities for optimization. While trying to overcome the second constraint creates a paradoxical situation of having to compute the next iterations of CORDIC to look ahead to those iterations in CORDIC reducing the number of instructions required by each iteration is actually quite doable. Each CORDIC iteration is effectively a string of predetermined arithmetic operations making the creation of a hardware assist rather straightforward.

4.4 Fast Parallelization of CORDIC with NEON

Implementing CORDIC with NEON intrinsics revealed an interesting opportunity to optimize CORDIC. ARM NEON intrinsics provide the opportunity to parallelize the CORDIC algorithm and compute multiple vectors at once in such a way that computing eight vectors is only marginally more computationally intensive than one.

Throughout this report the X and Y components have been stored in int32x2 NEON vectors. The NEON instruction set, however, supplies a number of other vector types with higher element counts [10]. Thus with minimal modification our implementation of CORDIC could be changed to compute two or more vectors instead of only one. There would be some performance degradation due to increased overhead such as a `sign` variable being computed for each vector. However, previously unmodified operations, such as the computation of Z (the rotation angle) could be parallelized. This parallelized version is likely to be massively faster than running the CORDIC algorithm multiple times sequentially

As a proof of concept the following is a for loop from an implementation of CORDIC which processes two vectors in parallel

```
// xy_neon is now int16x4_t

for(; i<15; i++) { /* 15 iterations are needed */

    xy_neon = vmla_s16(xy_neon, yx_neon, sign); // add and process signs

    sign[0] = xy_neon[1] > 0 ? 1 : -1; // compute sign for first vector
    sign[1] = -sign[0];
    sign[2] = xy_neon[3] > 0 ? 1 : -1; // compute sign for second vector
    sign[3] = -sign[2];
    yx_neon = vshr_n_s16(vrev32_s16(xy_neon), i); // copy and shift

    new_angle = vld1_s16(&z_table[i]); // retrieve the next elementary angle

    z_temp = vmla_s16(z_temp, new_angle, sign); // update the rotation angles

  }
```

A full version of the code can be found in appendix A.

Notice that with the only additional computations are required: computation of a second sign value and loading of the next elementary angle to allow the computation of Z to be parallelized. With only these two changes the assembly code remains relatively unchanged resulting in throughput of CORDIC being potentially doubled.

4.5 Performance Gain from Proposed Hardware Assist

Inspecting the assembly code generated from the C code containing the hardware assist implies that that hardware assist would likely lead to a significant performance improvement. Comparing the assembly code to that of the original implementation of CORDIC further confirms this:

One loop iteration of original implementation	One loop iteration with hardware assist

`cmp    r3, #0`
`ldr    lr, [r2, #36]`
`subgt  r3, r3, r4`
`addle  r3, r3, r4`
`addgt  r0, r0, r5`
`suble  r0, r0, r5`
`asr    r4, r0, #10`
`asr    r5, r3, #10`
`addgt  ip, ip, lr`
`suble  ip, ip, lr`
	`      cmp    r0, #0`
	`      mla    ip, lr, r2, ip`
	`      movgt  r2, #1`
	`      mvnle  r2, #0`
	`      mov    lr, #6`
	`      .syntax divided`
	`@ 17 "./cordic_V_fixed_point_hw.c" 1`
	`      cordic_hw    r0, r0, lr`
	
	`@ 0 "" 2`
	`      .arm`

	`.syntax unified`
	`ldr    lr, [r3, #24]`

The assembly contains only one arithmetic instruction (used for computing Z) compared to the five required by the original CORDIC implementation.

5. Conclusion

The project was not overly successful in achieving its initial goal of improving the performance of the CORDIC algorithm implemented in C. Despite this, it did prove worthwhile in other ways, namely, with the discovery that CORDIC might be very cheaply (from a performance standpoint) parallelized to handle multiple input vectors at once. Additionally, this report might provide some general guidance for anyone attempting to optimize an algorithm whose implementation is already fairly well optimized and/or short is length given the related challenges encountered by this project.

5.1 Future Work

Future work based on the work completed and lessons learned by this report might be focused in two areas: developing a CORDIC implementation using NEON intrinsics for processing multiple vectors in parallel and further development of a hardware assist. Given that our proposed hardware assist for CORDIC appears to provide a significant performance boost while remaining relatively simple we believe it may warrant future investigation. A study of the exact performance gain which might be expected would likely be a good starting point. From there, assuming the performance gain is considered sufficient, constructing an actual hardware prototype would not be overly difficult given the simplicity of the proposed design.

The code provided for parallelized CORDIC in section 4.5 serves only as a proof of concept. Should additional work be put into this idea optimizations are likely to be achieved. Additionally, the code might be further developed to process four or more vectors in parallel (note that NEON has support for int16x8 and even int16x8x2 vectors). From what was learned in this report we believe this path offers the best opportunity to significantly improve the performance of CORDIC using a software only approach (i.e. no firmware or hardware additions).

6. References

[1]	All About Circuits. *An introduction to the CORDIC Algorithm*. Accessed 2020. Available at: https://www.allaboutcircuits.com/technical-articles/ an-introduction-to-the-cordic-algorithm/ [Accessed: 14- Aug- 2020]

[2]	T. Vladimirova and H. Tiggeler, "FPGA Implementation of Sine and Cosine Generators Using the CORDIC Algorithm", *Citeseerx.ist.psu.edu*. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.581.4145&rep=rep1&type=pdf#:~:text=iz-,The%20CORDIC%20method%20can%20be%20employed%20in%20two%20different%20modes,a%20given%20angle%2C%20are%20computed. [Accessed: 14- Aug- 2020]
[3]	"ARMv7-M Architecture Reference Manual", *Web.eecs.umich.edu*, 2010. [Online]. Available: https://web.eecs.umich.edu/~prabal/teaching/eecs373-f10/readings/ARMv7-M_ARM.pdf. [Accessed: 14- Aug- 2020]
[4]	"Documentation – Arm Developer", *Developer.arm.com*. [Online]. Available: https://developer.arm.com/documentation/dui0489/g/arm-and-thumb-instructions/multiply-instructions/mul--mla--and-mls. [Accessed: 14- Aug- 2020]
[5]	"Crash Course: Software Pipelining", *Cs.unm.edu*. [Online]. Available: https://www.cs.unm.edu/~dccannon/cs341/week11/pipeline.pdf. [Accessed: 14- Aug- 2020]
[6]	"Arm NEON programming quick reference guide", *Community.arm.com*, 2020. [Online]. Available: https://community.arm.com/developer/tools-software/oss-platforms/b/android-blog/posts/arm-neon-programming-quick-reference. [Accessed: 14- Aug- 2020]
[7]	"Documentation – Arm Developer", *Developer.arm.com*. [Online]. Available: https://developer.arm.com/documentation/dui0472/m/using-neon-support. [Accessed: 14- Aug- 2020]
[8]	"Barrel-shifter (8 bit)", *Tams.informatik.uni-hamburg.de*. [Online]. Available: https://tams.informatik.uni-hamburg.de/applets/hades/webdemos/10-gates/60-barrel/shifter8.html. [Accessed: 14- Aug- 2020]
[9]	"Assembler User Guide: LDR pseudo-instruction", *Keil.com*. [Online]. Available: https://www.keil.com/support/man/docs/armasm/armasm_dom1361289875065.htm. [Accessed: 14- Aug- 2020]
[10]	[9]"Documentation – Arm Developer", *Developer.arm.com*. [Online]. Available: https://developer.arm.com/documentation/dui0473/m/neon-programming/neon-vectors. [Accessed: 14- Aug- 2020]

7. Appendices

Appendix A: C Code

Implementation of CORDIC using manual loop unrolling:

```
int cordic_V_fixed_point(int xy, int *z) {
  int x_temp_1, y_temp_1, z_temp;
  int x_temp_2, y_temp_2;
  int i;

  x_temp_1 = xy & 0xffff;
  y_temp_1 = (xy >> 16) & 0xffff;
  z_temp = 0;

  for( i=0; i<14; i+=2) { /* 15 iterations are needed */
       if( y_temp_1 > 0) {
       x_temp_2 = x_temp_1 + (y_temp_1 >> i);
       y_temp_2 = y_temp_1 - (x_temp_1 >> i);
       z_temp += z_table[i];
       } else {
       x_temp_2 = x_temp_1 - (y_temp_1 >> i);
       y_temp_2 = y_temp_1 + (x_temp_1 >> i);
       z_temp -= z_table[i];
       }
       x_temp_1 = x_temp_2;
       y_temp_1 = y_temp_2;

       if( y_temp_1 > 0) {
       x_temp_2 = x_temp_1 + (y_temp_1 >> i+1);
       y_temp_2 = y_temp_1 - (x_temp_1 >> i+1);
       z_temp += z_table[i+1];
       } else {
       x_temp_2 = x_temp_1 - (y_temp_1 >> i+1);
       y_temp_2 = y_temp_1 + (x_temp_1 >> i+1);
       z_temp -= z_table[i+1];
       }
       x_temp_1 = x_temp_2;
       y_temp_1 = y_temp_2;
  }

  if( y_temp_1 > 0) {
       x_temp_2 = x_temp_1 + (y_temp_1 >> i);
       y_temp_2 = y_temp_1 - (x_temp_1 >> i);
       z_temp += z_table[i];
  } else {
       x_temp_2 = x_temp_1 - (y_temp_1 >> i);
       y_temp_2 = y_temp_1 + (x_temp_1 >> i);
       z_temp -= z_table[i];
  }
  x_temp_1 = x_temp_2;
  y_temp_1 = y_temp_2;

  *z = z_temp;

  return y_temp_1 << 16 | x_temp_1;
}
```

Implementation of CORDIC which processes two vectors in parallel.

```
#include <stdio.h>
```

```
#include "arm_neon.h"

int16_t z_table[15];

int cordic_V_fixed_point(int *x1, int *y1, int *x2, int *y2, int *z1, int *z2) {
  int i = 1;
  int16x4_t sign;

  int16x4_t z_temp = {0,0,0,0};

  int16x4_t new_angle;

  int16x4_t xy_neon = {*x1, *y1, *x2, *y2};

  int16x4_t yx_neon = {*y1, *x1, *y2, *x2};

  sign[0] = xy_neon[1] > 0 ? 1 : -1; // get new sign

  sign[1] = xy_neon[3] > 0 ? 1 : -1; // get new sign

  new_angle = vld1_s16(&z_table[0]);

  z_temp = vmla_s16(z_temp, new_angle, sign);

  for(; i<15; i++) { /* 15 iterations are needed */

    xy_neon = vmla_s16(xy_neon, yx_neon, sign); // add

    sign[0] = xy_neon[1] > 0 ? 1 : -1; // get new sign
    sign[1] = -sign[0];
    sign[2] = xy_neon[3] > 0 ? 1 : -1; // get new sign
    sign[3] = -sign[2];
    yx_neon = vshr_n_s16(vrev32_s16(xy_neon), i); // copy and shift

    new_angle = vld1_s16(&z_table[i]);

    z_temp = vmla_s16(z_temp, new_angle, sign);

  }

  xy_neon = vmla_s16(xy_neon, yx_neon, sign); // add

  *x1 = xy_neon[0];
  *y1 = xy_neon[1];
  *x2 = xy_neon[2];
  *y2 = xy_neon[3];
  *z1 = z_temp[0];
  *z2 = z_temp[2];

  return 0;
}
```

To view any of the C code referenced in this report, please look at the supplied folder titled “C code”

Appendix B: Assembly Code

To view the assembly code referenced in this report, please look at the supplied folder titled “Assembly”.

Appendix C: Explanation of NEON Intrinsics

Intrinsic or Data Type	Explanation [7]
`int32_t`	A 32 bit integer
`int32x2_t`	Two 32 bit integers stored in a single vector-like object
`VSHR_N_S32`	Handles the the right bit shift on Y and X by i bits
`VMLA_N_S32`	Handles storing a second copy of X and Y in a single operation.
`VREV64_S32`	Handles the multiplication with the `sign` variable and the addition of X to Y and Y to x