University of Victoria
Faculty of Engineering
SENG 440 Embedded Systems: Final Project Report
Summer Term, 2022

QR Decomposition in C: Optimizing for ARM920T

By:

Lucas Antonsen, V00923982
Department of Computer Science, University of Victoria
lucasantonsen@uvic.ca

Ty Ellison, V00916580
Department of Software Engineering, University of Victoria
tyellison@uvic.ca

Date:

August 18, 2022

Abstract

QR decomposition is challenging to implement on an embedded system. In this report, we discuss implementing a QR decomposition program in C using Modified Gram-Schmidt Orthogonalization and optimizing it for use on an ARM920T processor. QR decomposition on the ARM920T is difficult given the large number of dependent vector and matrix operations and real number arithmetic without a dedicated floating point coprocessor. The decomposition was optimized by implementing loop unrolling, a cache-oblivious matrix transpose and other techniques. We generated an orthogonal matrix Q and upper triangular matrix R from an input matrix contained in a text file on the ARM920T. We also developed a fixed point arithmetic library compatible with the QR decomposition code base. The optimizations improve the program runtime slightly and the cache-oblivious matrix is shown to reduce the number of cache misses.

Introduction

In this project, QR decomposition is implemented on an ARM920T machine with the Modified Gram-Schmidt Orthogonalization algorithm. QR decomposition takes a matrix A and decomposes it into an orthogonal matrix Q and an upper triangular matrix R such that the product of QR is equivalent to A. Being able to efficiently compute a QR decomposition is important for a variety of problems. Perhaps most notably the algorithm is used to solve the linear least-squares problem [1], which is frequently encountered in robotics, computational science, optimization, and machine learning.

QR decomposition is difficult to compute on an embedded processor for two main reasons: 1) repeated dependent arithmetic operations and 2) real number representation. Repeated dependent arithmetic operations affect algorithm performance by reducing the amount of instructions which can be pipelined and typically involve reading and writing data from slow regions of memory (depending on the size of the input matrices/vectors). Real numbers may contain fractional components, and fractional arithmetic is challenging for embedded processors since many, like the ARM920T, do not possess a dedicated floating point unit. Without a dedicated floating point coprocessor, fractional numbers need to be converted into a representation the ALU can interpret and perform arithmetic on - this conversion is costly and avoidable.

To calculate the QR decomposition of a matrix A, a C program was developed to read in strings from a text file, convert the strings into numbers, store these numbers in a two dimensional array, perform QR decomposition via MGSO, and write the Q and R matrices to an output text file. The C program contains a tertiary QR function which calls primary and secondary vector and matrix routines (see Appendix A). The program also implements a square root function using the Newton-Raphson Method. Additionally, several routines were written to perform fixed point arithmetic. It is important to note that not all the code is compliant with the fixed point routines. At this moment, updating the QR routines to be compliant with the fixed point routines is the most pertinent improvement that can be made.

To optimize our program, we implemented the following optimizations:

- Loop unrolling
- For loop optimization
- Cache-oblivious matrix transpose
- Math macros
- Operator strength reduction
- Branch reduction
- Function inlining

- Custom assembly square root instruction
- Fixed point arithmetic

The optimizations shall be further described in the Design Challenge section below. The report will also cover the following topics in order:

- Background of QR Decomposition
- Design Challenge
- Base Code
- Optimizations
- Numerical Results
- Conclusion
- Future Work

Background

QR decomposition is used in a variety of problems, including the linear least squares problem [1], the QR eigenvalue algorithm [2] and finding the rank of a matrix [3]. The algorithm decomposes a matrix A into an orthogonal matrix Q and upper triangular matrix R (Fig. 1).

$$A = [e_1|e_2|\cdots|e_n]\begin{bmatrix} v_1 \cdot e_1 & v_2 \cdot e_1 & \cdots & v_n \cdot e_1 \\ 0 & v_2 \cdot e_2 & \cdots & v_n \cdot e_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & v_n \cdot e_n \end{bmatrix} = QR$$

Figure 1: QR decomposition of matrix A into orthogonal matrix Q and upper triangular matrix R.

There are a variety of algorithms for QR decomposition including Gram-Schmidt Orthogonalization (GSO), Householder Transformations [4], or Givens Rotations [5]. We chose to use the Modified Gram-Schmidt Orthogonalization (MGSO) for this project.

GSO is demonstrated in Fig. 2. for a matrix A = [v1, v2, …, vn].

$$
\begin{aligned}
\mathbf{u}_1 &= \mathbf{v}_1, & \mathbf{e}_1 &= \frac{\mathbf{u}_1}{\|\mathbf{u}_1\|} \\
\mathbf{u}_2 &= \mathbf{v}_2 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_2), & \mathbf{e}_2 &= \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} \\
\mathbf{u}_3 &= \mathbf{v}_3 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_3) - \operatorname{proj}_{\mathbf{u}_2}(\mathbf{v}_3), & \mathbf{e}_3 &= \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|} \\
\mathbf{u}_4 &= \mathbf{v}_4 - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_4) - \operatorname{proj}_{\mathbf{u}_2}(\mathbf{v}_4) - \operatorname{proj}_{\mathbf{u}_3}(\mathbf{v}_4), & \mathbf{e}_4 &= \frac{\mathbf{u}_4}{\|\mathbf{u}_4\|} \\
&\vdots & &\vdots \\
\mathbf{u}_k &= \mathbf{v}_k - \sum_{j=1}^{k-1} \operatorname{proj}_{\mathbf{u}_j}(\mathbf{v}_k), & \mathbf{e}_k &= \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}.
\end{aligned}
$$

Figure 2: The process for Gram-Schmidt Orthogonalization to create the matrices Q and R.

Classical GSO is numerically unstable and results in a Q matrix that is not exactly orthogonal. To reduce the orthogonality error in the decomposition process, MGSO was used. For a given vector u_k, MGSO calculates the orthogonal projection u_{k+1} by subtracting the projection of previous orthogonal vectors on the current version of u_k until each previous orthogonal vector has been subtracted [6].

$$
\mathbf{u}_k = \mathbf{v}_k - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_k) - \operatorname{proj}_{\mathbf{u}_2}(\mathbf{v}_k) - \cdots - \operatorname{proj}_{\mathbf{u}_{k-1}}(\mathbf{v}_k),
$$

Figure 3: GSO process for calculating u_k.

$$
\begin{aligned}
\mathbf{u}_k^{(1)} &= \mathbf{v}_k - \operatorname{proj}_{\mathbf{u}_1}(\mathbf{v}_k), \\
\mathbf{u}_k^{(2)} &= \mathbf{u}_k^{(1)} - \operatorname{proj}_{\mathbf{u}_2}\left(\mathbf{u}_k^{(1)}\right), \\
&\vdots \\
\mathbf{u}_k^{(k-2)} &= \mathbf{u}_k^{(k-3)} - \operatorname{proj}_{\mathbf{u}_{k-2}}\left(\mathbf{u}_k^{(k-3)}\right), \\
\mathbf{u}_k^{(k-1)} &= \mathbf{u}_k^{(k-2)} - \operatorname{proj}_{\mathbf{u}_{k-1}}\left(\mathbf{u}_k^{(k-2)}\right), \\
\mathbf{e}_k &= \frac{\mathbf{u}_k^{(k-1)}}{\left\|\mathbf{u}_k^{(k-1)}\right\|}
\end{aligned}
$$

Figure 4: MGSO process.

MGSO results in a significant bump in stability, evident in Fig. 5 for GSO and MGSO applied to a Hilbert matrix. The test bench for this project used a 20x20 real valued matrix, which corresponds to a ~10^{10} decrease in orthogonality error [6]. This is why MGSO was selected instead of GSO.

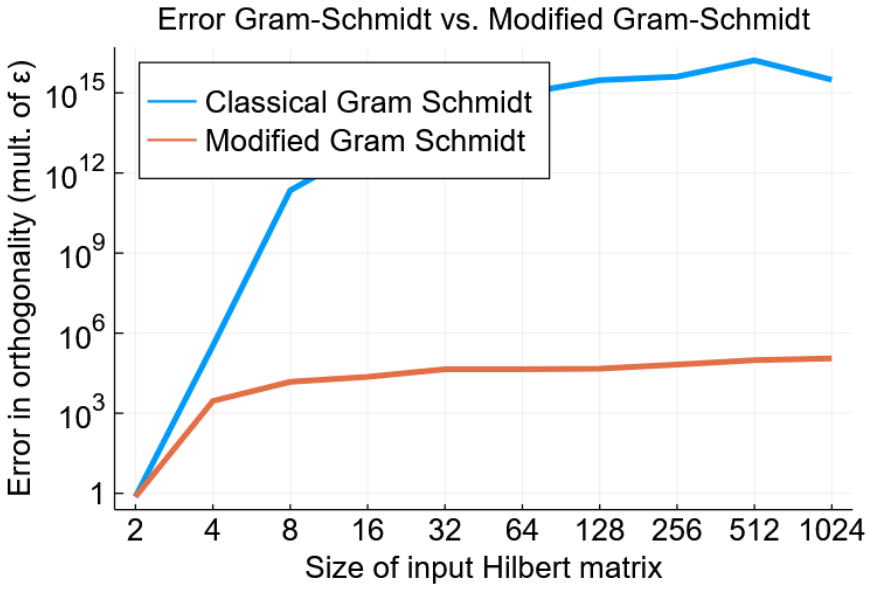
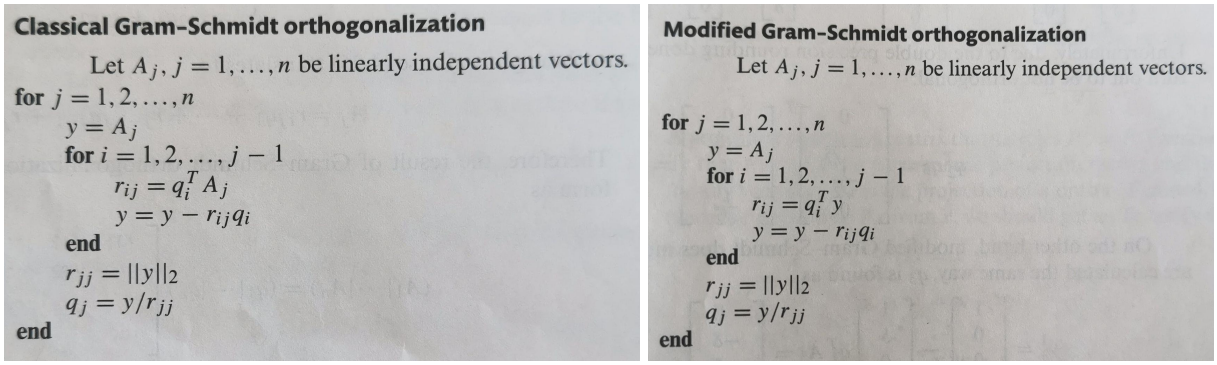


Figure 5: Orthogonality error for GSO vs MGSO on a Hilbert matrix [6].

To implement MGSO the following pseudocode as seen in Fig. 6 was used. This pseudocode influenced the design of our QR routine, as it directly outlines which kind of mathematical operations are required to perform QR decomposition: vector dot product, vector L2-norm, division of a vector by a constant, and element wise vector subtraction.



Classical Gram-Schmidt orthogonalization

Let $A_j, j = 1, \ldots, n$ be linearly independent vectors.

for $j = 1, 2, \ldots, n$
  $y = A_j$
  for $i = 1, 2, \ldots, j-1$
    $r_{ij} = q_i^T A_j$
    $y = y - r_{ij} q_i$
  end
  $r_{jj} = ||y||_2$
  $q_j = y / r_{jj}$
end

Modified Gram-Schmidt orthogonalization

Let $A_j, j = 1, \ldots, n$ be linearly independent vectors.

for $j = 1, 2, \ldots, n$
  $y = A_j$
  for $i = 1, 2, \ldots, j-1$
    $r_{ij} = q_i^T y$
    $y = y - r_{ij} q_i$
  end
  $r_{jj} = ||y||_2$
  $q_j = y / r_{jj}$
end

Figure 6: Pseudocode for the Classical (left) and Modified (right) Gram-Schmidt Orthogonalization algorithm [7].

Design Challenge

As previously stated, the greatest challenges in this project were performing 1) repeated dependent arithmetic operations and 2) performing arithmetic with real numbers. To build an efficient QR implementation several subroutines were required to handle file IO, compute numerical approximations, perform vector routines, and represent real numbers using integer types. File IO was implemented entirely through the stdio.h, stdlib.h, and string.h libraries. Numerical analysis algorithms were selected from [7] and basic vector routines were created to support the necessary operations. Once algorithms were selected and implemented, the code was then optimized to improve performance. To represent real numbers with integers, a fixed point library was developed to convert numerical strings directly to integers.

The MGSO algorithm requires the dot product, element wise vector subtraction, element wise vector multiplication, the L2-norm, and vector division by a constant. Simple algorithms exist to compute each of these operations and can be computed in linear time (O(n) where n represents the cardinality of a vector). Since these simple algorithms converge in linear time, and the size of the test matrices are such that introducing parallelism may not necessarily contribute to a large boost in performance, it was decided that these simple sequential algorithms would be sufficient. For example, consider the L2-norm of a vector with *N* elements (Eq. 1).

$$|v|_2 = \sqrt{\sum_{k=1}^{N} v_k^2} \tag{1}$$

To compute the L2 norm of v, we first dot product v with itself ($v^T \bullet v$) and then calculate the square root of this. However, to be compliant with the future fixed point standard (i.e. equation constants can be converted to their appropriate values for any given fixed point scale factor, and the return type can be explicitly set to an integer value), the square root function was implemented in software using the Newton-Raphson method for calculating roots to algebraic equations (Eq. 2). The Newton-Raphson method is defined as

$$x_n = x_{n-1} - \frac{f_{n-1}}{\dot{f}_{n-1}} \tag{2}$$

Consider the following function where a is an arbitrary constant,

$$f(x) = x^2 - a \tag{3}$$

The derivative of this function is simply

$$f'(x) = 2x \tag{4}$$

Let us define the value of x such that $f(x) = 0$ as x_* and choose a to be an initial guess for x_*. The Newton-Raphson exhibits quadratic convergence towards the root x_* given a is sufficiently close to x_*. Given the goal is to compute a square root, we can choose a intelligently to be the closest integer which when squared yields the floor of x simply by starting with some initial guess, squaring it, checking to see if the square is greater than the integer component of the rooted value, and if not then increment the guess and repeat. By choosing a this way, the problem of finding a close enough root is reduced to repeated integer multiplication and addition operations, and reduces the number of iterations required by the Newton-Raphson method. Substituting Eq. 3 and Eq. 4 into Eq. 2 we get:

$$x_n = x_{n-1} - \frac{x_{n-1}^2 - a}{2x_{n-1}} \tag{5}$$

It is important to note that the Newton-Raphson method is not guaranteed to converge to the correct root, but provided the algorithm has a sufficiently good initial guess, a reasonable error tolerance, and a max number of iterations, convergence is highly likely. Since the ARM920T has 32 bit registers and all of the numbers we decided to choose for testing fall in the range of signed 32 bit integers, we set the maximum number of iterations to be 46,340 as this is close to the square root of the max 32 bit integer. The convergence tolerance (difference in relative forward error) was set to be 10^{-14} as this is close to machine epsilon, and a reference value for machine

epsilon was set to 10^{-15} so no algorithm ever made comparisons on the order of exact machine epsilon.

The fixed point arithmetic library utilized 32 bit signed and unsigned integers. The numbers were chosen to be represented in Two's Complement with 1 sign bit, 16 whole bits, and 15 fractional bits. This system can represent numbers in the range [-65,536.99997, 65,536.99997] with a resolution of $3 \cdot 10^{-5}$ and truncation rounding scheme. Truncation was chosen instead of a more robust rounding scheme because the precision of the current arithmetic system is sufficiently high and the dimensionality of the matrices is sufficiently small such that truncation error will not accumulate to a point where it destroys the accuracy of the decomposition. The algorithm avoids using intermediary floating point values by converting strings directly into their fixed point representations. To convert decimal fractions into binary fractions, a custom algorithm was used based heavily on the following scheme. Consider an example where we want to convert 0.0432 into a binary fraction, we begin by multiplying the number by 2 to get 0.0864 and checking to see if the result is greater than 1. Since the result is less than 1, we record a 0 for the first bit of the sequence and repeat the previous steps until all bits in the finite sequence have been occupied. If the result of the multiplication is greater than 1, we record a 1 in the bit sequence and then subtract 1 from the result.

$$
\begin{aligned}
0.0432 \cdot 2 &= 0.0864 && 0\\
0.0864 \cdot 2 &= 0.1728 && 0\\
0.1728 \cdot 2 &= 0.3456 && 0\\
0.3456 \cdot 2 &= 0.6912 && 0\\
0.6912 \cdot 2 &= 1.3824 && 1\\
0.3824 \cdot 2 &= 0.7648 && 0\\
0.7648 \cdot 2 &= 1.5296 && 1\\
0.5296 \cdot 2 &= 1.0592 && 1\\
0.0592 \cdot 2 &= 0.1184 && 0\\
&\vdots
\end{aligned}
$$

Since integer values are used to represent fractional numbers, the algorithm is modified to replace the greater than 1 comparison to a comparison with a threshold integer value which is $10^{|d|}$ where |d| represents the number of significant digits in the decimal fraction. Additionally, the fractional values are treated as whole numbers (e.g. 0.0432 is treated as 432). An example for the revised algorithm with the same initial value as before:

$$
\begin{aligned}
\text{threshold} &= 10^4 &&\\
432 \cdot 2 &= 864 && 0
\end{aligned}
$$

$$
\begin{aligned}
864 \cdot 2 &= 1\,728 && 0\\
1728 \cdot 2 &= 3\,456 && 0\\
3456 \cdot 2 &= 6\,912 && 0\\
6912 \cdot 2 &= 13\,824 && 1\\
3824 \cdot 2 &= 7\,648 && 0\\
7648 \cdot 2 &= 15\,296 && 1\\
5296 \cdot 2 &= 10\,592 && 1\\
592 \cdot 2 &= 1\,184 && 0\\
&\vdots
\end{aligned}
$$

Sequential processors have much stricter hardware limitations in comparison to regular computers, thus when approaching this problem it is necessary to investigate ways to optimize the C code such that the program runs as efficiently as possible.

Concerning MGSO, the program requires many function calls to handle the various vector/matrix operations. Function calls are costly on program efficiency but cannot be changed since the function calls are primarily for loops and would create confusion if they were added to the QR function. To reduce the cost of these functions, their for loops will need optimization.

Since the ARM920T processors do not natively support floating point arithmetic (require an additional coprocessor), it will be necessary to implement a dedicated fixed point arithmetic library to improve. Math routines for vectors and matrices are not available through library support so functions will need to be created.

Code Optimization Plan

We intend to implement the following optimizations:

- Loop unrolling
- Cache-oblivious matrix transpose
- Macros
- Operator strength reduction
- Inline functions
- Fixed Point Arithmetic
- Reduce branch and load/store instructions
- Dedicated square root instruction

Base Code - (See Code Submitted)

The base code is composed of three files, qr.c, qr.h, and fixed.h.

qr.c is the driver code used for running the QR decomposition and is responsible for initializing all the matrices and variables and calling the functions to prepare, run and output the results of MGSO. qr.c also measures the run-time of the program using time.h.

qr.h contains the QR function for computing the QR decomposition via MGSO and the supporting vector functions.

The supporting functions and their roles are described in the Appendix under Supporting Function Description.

Optimizations:

Optimization 1: Loop Unrolling

Given that many of our functions are primarily for loops it makes sense to implement software pipelining or loop unrolling on these functions to improve efficiency. We implemented loop unrolling as we found it too challenging to implement software pipelining. A loop unrolled function is found below in Fig. 7.

We chose to unroll the loop only once as this limits the pressure on the register file. We added the if statement in general before performing the next operation of the loop unroll as we needed to ensure a null pointer exception would not occur when indexing an odd sized matrix or vector.

```
//loop unrolling done
void numt_set_col(SIZE_T rows, SIZE_T cols, SIZE_T target_col, NUM_T v[rows], NUM_T A[rows][cols]) {
    SIZE_T i;

    for (i=0; i<rows; i+=2) {
        A[i][target_col] = (NUM_T) v[i];

                if(i+1 != rows){
                        A[i+1][target_col] = (NUM_T) v[i+1];
                }
    }
}
```

Figure 7: Example of loop unrolling in our program.

Optimization 2: For Loop Optimized

In the functions sqr_rt and closest_perfect_square we altered the for loop to count down from the max iterations (Fig. 8.) rather than to count up as it is cheaper to check the zero flag as the exit condition than to do a comparison between i and the max_iter variable. A normal i < max_iter exit condition takes 1 cycle for subtracting max_iter and 1 cycle to test the result (zero flag). Our optimized version takes only 1 cycle to check the zero flag.

```
for (i=max_iter; i!=0; i-=2) {
```

Figure 8: For loop optimization.

Optimization 3: Cache-oblivious Matrix Transpose

To attempt to reduce the number of cache misses of our matrix transpose found in Fig. 9, we implemented a cache-oblivious matrix transpose (Fig. 10.). The primary issue with our initial matrix transpose is that data is stored in row-major order. When the matrix is brought into the cache, a row is brought in rather than a column. This means we will consistently miss as we try to access values down a column and each time will have to load another row into the cache.

```
void transpose_m(SIZE_T rows, SIZE_T cols, DATA_T A[rows][cols], DATA_T B[cols][rows]) {
        SIZE_T i, j;
        for (i=0; i<cols; ++i) {
                for (j=0; j<rows; ++j) {
                        B[i][j] = A[j][i];
                }
        }
}
```

Figure 9: Initial unoptimized matrix transpose.

```
//cache friendly matrix transpose
void transpose_m(SIZE_T rows, SIZE_T cols, DATA_T A[rows][cols], DATA_T B[cols][rows], SIZE_T start_i, SIZE_T end_i, SIZE_T start_j, SIZE_T end_j){

        SIZE_T l_i = end_i - start_i;
        SIZE_T l_j = end_j - start_j;
        SIZE_T i, j;

        if (l_i <= 2 && l_j <= 2) {
                for (i = start_i; i < end_i; i++) {
                        for (j = start_j; j < end_j; j++) {
                                B[j][i] = A[i][j];
                        }
                }
        } else if (l_i >= l_j) {
                transpose_m(rows, cols, A, B, start_i, start_i + (l_i / 2), start_j, end_j);
                transpose_m(rows, cols, A, B, start_i + (l_i / 2), end_i, start_j, end_j);
        } else {
                transpose_m(rows, cols, A, B, start_i, end_i, start_j, start_j + (l_j / 2));
                transpose_m(rows, cols, A, B, start_i, end_i, start_j + (l_j / 2), end_j);
        }
}
```

Figure 10: Cache-oblivious matrix transpose.

The cache-oblivious matrix transpose splits the input matrix into smaller and smaller sub-matrices either by length or width until the length and width are less than 2 (sub-matrix size <= 4) and then performs the matrix transpose on the sub-matrix. By implementing the cache-oblivious matrix transpose, we reduce the number of cache misses as we operate on smaller chunks of the matrix (and memory), reducing the number of rows that need to be brought into cache. Fig. 11. demonstrates the matrix transpose operation.

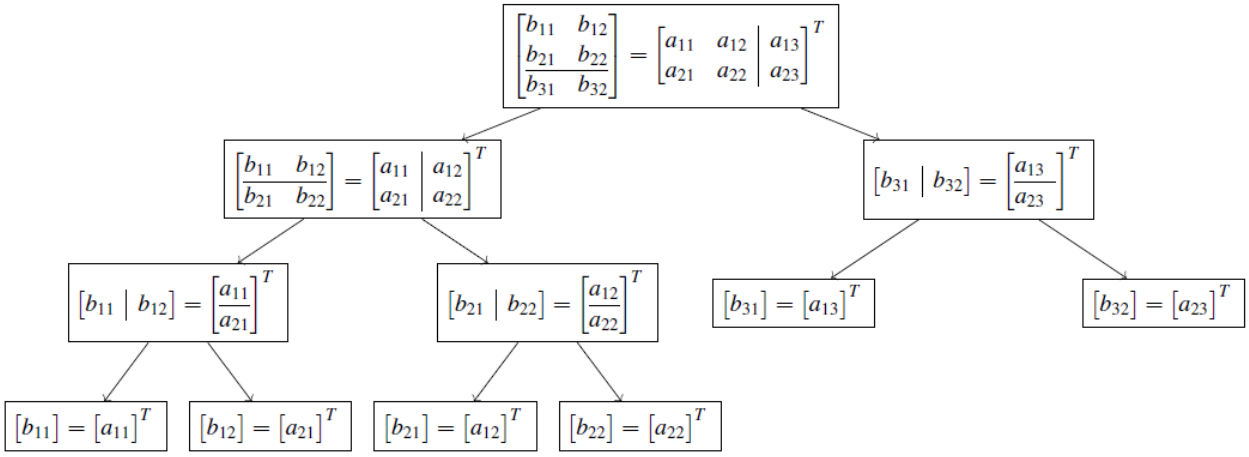


Figure 11: Cache-oblivious matrix transpose in action. Image from SENG 475 Lecture Slides by Michael Adams.

Optimization 4: Macros

We implemented a f_n macro for computing the value of f(x) and a compute x_n for computing the value of x_n for the Newton-Raphson method (sqr_rt function) (Fig. 12). We implemented a compute_l macro for calculating the length of a matrix chunk in transpose_m (Fig. 12).

```
#define compute_f_n(x_0,x_) (x_0)*(x_0) - x_
#define compute_xn(x_0,fn,fprime) x_0 - (fn / fprime)
#define compute_l(end,start) end - start
```

Figure 12: Macros for qr.h.

Optimization 5: Operator Strength Reduction

To reduce the cost of operators, we removed the division from the for loop in vec_divc and replaced it with multiplying by the constant div = 1/divisor (Fig. 12.). By multiplying instead of dividing, we go from 10 cycles to 3 cycles. Similarly, in transpose_m we replaced division by 2 for dividing the matrix length to shift left, going from 10 cycles to 1 cycle.

```
NUM_T div = 1/divisor;  //operator strength reduction

for (i=0; i<size-1; i+=2){
        v[i] *= div;
```

Figure 12: Operator strength reduction in div_c.

Optimization 6: Branch Reduction

We removed the if statements within the loop unrolled functions (Fig. 7.) and instead checked if the size is even or odd outside the loop, and updated the matrix/vector if odd (Fig. 13.). We iterate through i equals 0 to size-1 so that we avoid the null pointer for the last odd entry if it exists.

```
for (i=0; i<size-1; i+=2) {
        v[i] *= c;
        v[i+1] *= c;
}
if(size & 1){
        v[size-1] *= c;
}
```

Figure 13: Branch reduction in loop unrolled code.

Optimization 7: Inline Functions

Function inlining hints to the compiler to extract the body of the inline function and place it directly in the routine which calls it. Since both our fixed point and QR routines rely on calling functions within functions, inlining can be used to reduce call-linkage overhead. As a general guideline, it is recommended for functions to be inlined when the function definition is small, the function is called repeatedly, and the function is not IO bound. An example of this would be the function we use to find the binary representation of a decimal fraction. This function is called each time a decimal number which contains a fractional component is read in from a text file as a string and converted into a fixed point number.

```
UFX_T str_to_fx(char *s, char *delim, FX_SIZE_T scale) {
    FX_SIZE_T sign = 0;
    FX_T num = 0;
    FX_SIZE_T digits = 0;
    UFX_T threshold = 0;

    char *p;
    char *tok = strtok(s, delim);
    num = (FX_T) strtol(tok, &p, 10);
    if (num < 0) {
        sign = 1;
        num -= 1;
        num = ~num;
    }
    num = ((UFX_T) num << scale);

    tok = strtok(NULL, delim);
    if (tok != NULL) {
        digits = strnlen(tok, FX_MAX_DEC_CHARS);
        threshold = get_threshold(tok, digits);
        num |= fract_dec_to_bin((UFX_T) strtoul(tok, &p, 10), threshold);
    }
    return (sign) ? FX_SIGN | num : num;
}
```

```
inline UFX_T fract_dec_to_bin(UFX_T x, UFX_T threshold) {
    UFX_T fract_bin = 0;
    FX_SIZE_T i = 0;

    for (; i < FX_FRACT_BITS; ++i) {
        fract_bin <<= 1;
        if ((x <<= 1) > threshold) {
            fract_bin |= 1;
            x -= threshold;
        }
    }
    return fract_bin;
}
```

Figure 14: String to fixed point conversion routine which calls inlined fract_dec_to_bin if the decimal number contains a fractional component.

Optimization 8: Fixed Point Arithmetic

Fixed point arithmetic routines were developed to eliminate floating point data types. For real time applications such as implementing a least squares heuristic in A* search for robotic path planning, it is imperative that arithmetic operations are fast. Relying on software libraries to convert floats into alternative representations adds a computational overhead that can be avoided by choosing to represent numbers in fixed point.

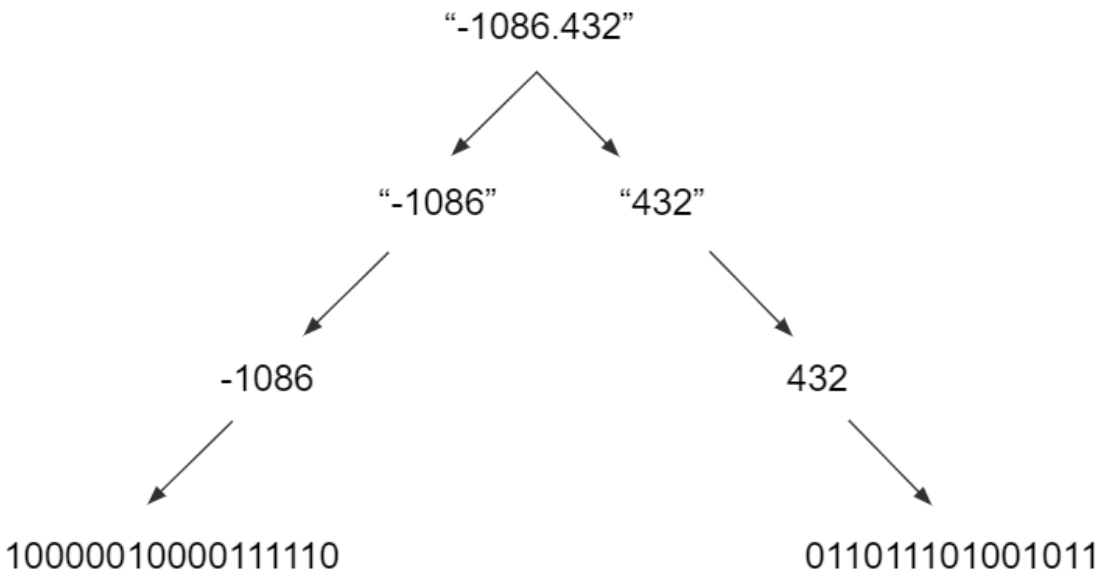


Figure 15: Visual representation of how our program splits a decimal number into its whole and fractional components and then converts the decimal fraction into a binary fraction before combining the binary digits.

Optimization 9: Dedicated Square Root Instruction

Every row of the input matrix is used to initialize the y-vector for orthogonalization. Since each entry in the R matrix is the L2-norm of the orthogonalized y-vector, the L2-norm must be calculated for each row in the input matrix. Calculating a square root purely in software is slow as it relies on repeated branching, type casting, and nested function calls to check if termination conditions have been satisfied. More specifically, there were 24 branch operations (3 cycles), and 10 load and store operations (4 cycles) in the assembly code for this routine alone. Since the number of calls to the square root routine is linearly dependent on the number of rows in the input matrix A, we can define a function to calculate the estimated number of cycles saved by removing these instructions. Let us define the following function $f(\bullet)$, where r is the number of rows of the input matrix A, b is the number of branch instructions, and m is the number of load and store operations. The coefficients for b and m represent the number of cycles each instruction costs.

$$f(r) = r \cdot [3b + 4m] \tag{6}$$

Given there were 24 branch instructions and 10 load and store instructions in the assembly code generated by the square root routine, we can set $b = 24$ and $m = 10$.

$$f(r) = r \cdot [3(24) + 4(10)]$$

$$f(r) = 112r$$

Let us consider the 20 by 20 input matrix utilized by our test suite, then $r = 20$. We can now use Eq. 6 to calculate the number of cycles spent just moving to other locations in the code and

loading values to and from memory.

$$f(20) = 112(20)$$

$$f(20) = 2240$$

In addition to removing excess branch, load, and store operations, the entire square root routine can be replaced with a dedicated square root instruction. Two input registers which represent the source register with the value to be rooted, and the destination register where the ALU will place the result (see Fig. 16). Each square root instruction represents an inline function call to the ALU. Replacing the software implementation of the square root with a dedicated instruction will remove 130 lines from the assembly code, which translates to an 8% reduction in code size.

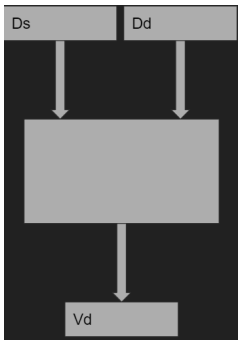


Figure 16: Register representation of square root instruction, Ds is the source value, Dd is the destination register, Vd represents the computed square root of Ds.

Assembly Comparison:

Loop unrolling was performed on the majority of our program's functions and resulted in an increase in assembly code from the base code in Fig. 18 (Appendix) to the optimized code in Fig. 19 (Appendix). The loop unrolled code increased in size due to the new vector update for i+1. The added if statement for avoiding null pointers for odd sized vectors further increased the code size and added 2 additional branches.

Compilation and Run Commands

To compile the program:
arm-linux-gcc -static -O2 qr.c -o qr.exe

To run the program:
./qr.exe input_matrix.txt qr_output.txt matrix_rows matrix_columns

The -O2 flag notifies the compiler to turn on a specific set of optimization flags. -O2 is a less aggressive flag than -O3.

Numerical Results

Running our program with a 20 by 20 matrix (m20x20.txt) 100 times on the ARM920T we observed the program runtime measurements below in Table 1 for using time.h.

Note: We do not include fixed point arithmetic data as it was not integrated into qr.h. Inlining data is not included as it is in the fixed point arithmetic code.

Table 1: Performance results for qr.h in order of optimization implemented.

QR Version	**Runtime (seconds)**	**Difference From Previous Version (seconds)**
Base	0.739600	-
Loop Unrolled	0.736900	0.0027
For Loop Optimized	0.736600	0.0003
Cache-oblivious Matrix Transpose	0.736400	0.0002
Macros	0.736600	-0.0002
Operator Strength Reduction	0.738400	-0.0018
Branch Reduction	0.745700	-0.0073

Given the number of function calls, the level of improvement was limited as there was no way to avoid them. The most significant improvement was the loop unrolling which impacted the majority of the functions in the program, but the improvement was small, only 0.0027 seconds. The program performance plateaued after the loop unrolling and hit the minimum of 0.736400 seconds with the cache-oblivious matrix transpose. The program performance decreased significantly after operator strength reduction with a runtime of 0.738400 seconds and became worse than the base version with a runtime of 0.745700 seconds after branch reduction was implemented.

We measured the number of cache misses for one run of each version of the matrix transpose using a 20 by 20 matrix (m20x20.txt) with cachegrind (Fig. 17). To perform this test, only the function calls necessary for matrix transpose to run were used in qr.c and all other calls were commented out. The cache oblivious matrix transpose showed improvements in cache misses but not in branches. I references increased by 23,079 but cache misses for I1 and LLi decreased by

0.01% and 0.02% respectively. D references increased by 15,072 but caches misses for D1 and LLd decreased by 0.2%. LL references increased by 10 but cache misses for LL decreased by 0.1%. Branches increased by 1,221 and mispredictions increased by 0.1%.

```
I   refs:      407,515
I1  misses:      1,013
LLi misses:      1,007
I1  miss rate:    0.25%
LLi miss rate:    0.25%

D   refs:      141,806  (89,773 rd   + 52,033 wr)
D1  misses:      3,318  ( 2,518 rd   +    800 wr)
LLd misses:      2,628  ( 1,995 rd   +    633 wr)
D1  miss rate:     2.3% (   2.8%     +    1.5%  )
LLd miss rate:     1.9% (   2.2%     +    1.2%  )

LL refs:         4,331  ( 3,531 rd   +    800 wr)
LL misses:       3,635  ( 3,002 rd   +    633 wr)
LL miss rate:      0.7% (   0.6%     +    1.2%  )

Branches:       79,195  (78,038 cond +  1,157 ind)
Mispredicts:     5,700  ( 5,603 cond +     97 ind)
Mispred rate:      7.2% (   7.2%     +    8.4%   )
```

```
I   refs:      430,594
I1  misses:      1,017
LLi misses:      1,009
I1  miss rate:    0.24%
LLi miss rate:    0.23%

D   refs:      156,878  (98,684 rd   + 58,194 wr)
D1  misses:      3,324  ( 2,517 rd   +    807 wr)
LLd misses:      2,630  ( 1,994 rd   +    636 wr)
D1  miss rate:     2.1% (   2.6%     +    1.4%  )
LLd miss rate:     1.7% (   2.0%     +    1.1%  )

LL refs:         4,341  ( 3,534 rd   +    807 wr)
LL misses:       3,639  ( 3,003 rd   +    636 wr)
LL miss rate:      0.6% (   0.6%     +    1.1%  )

Branches:       80,416  (79,259 cond +  1,157 ind)
Mispredicts:     5,878  ( 5,781 cond +     97 ind)
Mispred rate:      7.3% (   7.3%     +    8.4%   )
```

Figure 17: Cachegrind output for base (left) and cache-oblivious matrix transpose (right).

Conclusion

QR decomposition is challenging to implement and optimize on an embedded system due to the requirements of many functions for matrix and vector operations. Matrix and vector operations require many simple for loops which are difficult to optimize other than with loop unrolling. The -O2 compiler flag optimizes the code significantly and a cache-oblivious matrix transpose lowers cache misses but does not have a strong effect on runtime. Fixed point arithmetic for QR decomposition is challenging to implement on an embedded machine.

Future Work

In the future, we would like to implement an MLA instruction to parallelize vector and matrix routines and implement more robust rounding schemes for decimal to fixed point conversion to reduce round-off error. We would also like to integrate the existing QR decomposition code with the custom fixed point routines and develop more robust testing procedures which measure round-off and truncation error, and runtime for matrices of arbitrary size and conditioning. Researching other QR decomposition algorithms and implementing them on an embedded system would also be useful.

References

[1] B. D. Shaffer. “QR Matrix Factorization.” towardsdatascience.com. https://towardsdatascience.com/qr-matrix-factorization-15bae43a6b2 (accessed Feb. 27, 2022).

[2] J. W. Demmel, *Applied Numerical Linear Algebra*. SIAM, 1997.

[3] T. F. Chan, “Rank revealing QR factorizations,” *Linear Algebra and its Applications*, vol. 88–89, pp 67-82, 1987, doi: 10.1016/0024-3795(87)90103-0.

[4] C. Moler. “Householder Reflections and the QR Decomposition.” mathworks.com. https://blogs.mathworks.com/cleve/2016/10/03/householder-reflections-and-the-qr-decomposition/ (accessed Aug. 18, 2022).

[5] M. Fenner, “Givens Rotations and QR.” drfenner.org. http://drsfenner.org/blog/2016/03/givens-rotations-and-qr/ (accessed Aug. 18, 2022).

[6] L. Hoeltgen, “Gram-Schmidt vs. Modified Gram-Schmidt.” laurenthoeltgen.name. https://www.laurenthoeltgen.name/post/gram-schmidt/ (accessed Aug. 18, 2022)

[7] T. Sauer, *Numerical Analysis*, 3rd ed. Hoboken, NJ?: Pearson, 2018.

Appendix A

Supporting Function Description:

l2_norm: Calculates the magnitude of a vector using the Euclidean norm.

sqr_rt: Calculates the square root of a number via the Newton-Raphson method.

closest_perfect_square: Calculates the integer which when squared is closest to the input number.

abs_val: Calculates the absolute value of a given number.

numt_copy_col: Copies a column vector of a given matrix into another vector.

numt_set_col: Sets the column of a given matrix to the elements from the given vector.

mat_set_col: Sets the column of a given matrix to the elements from the given vector.

vec_sub: Subtracts one vector from another vector.

numt_vec_copy: Copies a vector to another vector.

vec_copy: Copies a vector to another vector.

vec_dot: Performs the dot product of two vectors.

vec_mulc: Multiplies a given vector by a constant.

vec_mul: Multiplies two vectors source and destination, and stores the result in the destination.

vec_divc: Divides a vector by a constant.

transpose_m: Computes the matrix transpose of matrix A and stores it in matrix B.

fscanm: Converts the matrix from the input file into a 2D array.

fprintmt: Outputs matrix to a file.

fprintm: Outputs matrix to a file.

printm: Outputs matrix to standard output.

printv: Outputs vector to standard output.

openf: Opens file and returns file pointer.

spec_map: Maps the input data type to its corresponding type specifier.

zero_m: Zeroes out matrix to avoid garbage values.

Appendix B

File Submission Description:

data folder: Contains matrix input files m0.txt to m4.txt and m20x20.txt.

qr_version folder: Contains the various versions of qr from the base version to the optimized versions. Each version is named to reflect its most recent optimization and corresponds to the results in Table 1.

README.md: Progress report.

fixed.h: Code for the fixed point arithmetic.

new_statistics.txt: Contains measurements of program performance through various stages of optimization.

qr-final_loop_unroll.s: Assembly code for the program after branch reduction is implemented. Corresponds to the final version of qr.h.

qr.c: Handles the invocation of the various functions to enable QR decomposition to occur.

qr.h: Contains the code for all the functions necessary for QR decomposition. Final version.

qr_base.s: Assembly code for the base version of the program.

qr_loop_unrolled.s: Assembly code for the loop unrolled version of the program.

transpose_no_opt_cache_stats.txt: Cachegrind output for the unoptimized transpose_m function.

transpose_opt_cache_stats.txt: Cachegrind output for the optimized transpose_m function.

Appendix C

vec_sub Assembly (Base and Loop Unrolled):

```
vec_sub:
        @ Function supports interworking.
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        cmp     r0, #0
        stmfd   sp!, {r4, r5, r6, r7, r8, lr}
        mov     r7, r1
        beq     .L31
        sub     r3, r0, #1
        mov     r3, r3, asl #16
        mov     r3, r3, lsr #13
        mov     r5, r2
        add     r6, r3, #8
        mov     r4, #0
.L30:
        add     r1, r7, r4
        ldmia   r1, {r2-r3}
        ldmia   r5, {r0-r1}
        bl      __aeabi_dsub
        add     r4, r4, #8
        cmp     r4, r6
        stmia   r5!, {r0-r1}
        bne     .L30
.L31:
        ldmfd   sp!, {r4, r5, r6, r7, r8, lr}
        bx      lr
        .size   vec_sub, .-vec_sub
        .align  2
        .global numt_vec_copy
        .type   numt_vec_copy, %function
```

Figure 18: vec_sub base assembly.

```
vec_sub:
        @ Function supports interworking.
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        stmfd   sp!, {r4, r5, r6, r7, r8, r9, sl, lr}
        subs    sl, r0, #0
        mov     r8, r1
        mov     r7, r2
        beq     .L36
        mov     r5, #0
.L35:
        mov     r4, r5, asl #3
        add     r1, r8, r4
        add     r4, r7, r4
        ldmia   r1, {r2-r3}
        ldmia   r4, {r0-r1}
        bl      __aeabi_dsub
        add     r2, r5, #1
        mov     r3, r2, asl #3
        cmp     r2, sl
        add     ip, r8, r3
        add     r6, r7, r3
        stmia   r4, {r0-r1}
        beq     .L34
        ldmia   ip, {r2-r3}
        ldmia   r6, {r0-r1}
        bl      __aeabi_dsub
        stmia   r6, {r0-r1}
.L34:
        add     r3, r5, #2
        mov     r3, r3, asl #16
        mov     r5, r3, lsr #16
        cmp     sl, r5
        bhi     .L35
```

```
.L36:
        ldmfd   sp!, {r4, r5, r6, r7, r8, r9, sl, lr}
        bx      lr
        .size   vec_sub, .-vec_sub
        .align  2
        .global numt_vec_copy
        .type   numt_vec_copy, %function
```

Figure 19: vec_sub loop unrolled assembly.