QR Decomposition on ARM920T

By Lucas Antonsen and Ty Ellison

Outline

Introduction to QR Decomposition

Design Challenges

Work Plan

Optimizations

Assembly

Numerical Results

Conclusion

What is QR Decomposition?

$$\overset{A}{\begin{pmatrix} 2.5 & 1.1 & 0.3 & 2.2 \\ 1.9 & 0.4 & 1.8 & 0.1 \\ 0.3 & 0.3 & 0.2 & 1.6 \\ 1.1 & 1. & 0.2 & 1.2 \end{pmatrix}} = \overset{Q}{\begin{pmatrix} -0.7 & 0.1 & 0.7 & -0.1 \\ -0.6 & -0.6 & -0.6 & 0.1 \\ -0.1 & 0.3 & -0.3 & -0.9 \\ -0.3 & 0.8 & -0.4 & 0.4 \end{pmatrix}} \overset{R}{\begin{pmatrix} -3.3 & -1.4 & -1.3 & -2.2 \\ 0. & 0.7 & -0.8 & 1.4 \\ 0. & 0. & -1. & 0.4 \\ 0. & 0. & 0. & -1.3 \end{pmatrix}}$$

Classical Gram-Schmidt Orthogonalization

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

Classical Gram-Schmidt Orthogonalization

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

Modified Gram-Schmidt Orthogonalization

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

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

CGSO vs. MGSO

Classical Gram–Schmidt orthogonalization

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

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

Modified Gram–Schmidt orthogonalization

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

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

Images reproduced from the third edition of “Numerical Analysis,” by Timothy Sauer.

Stability GSO vs MGSO

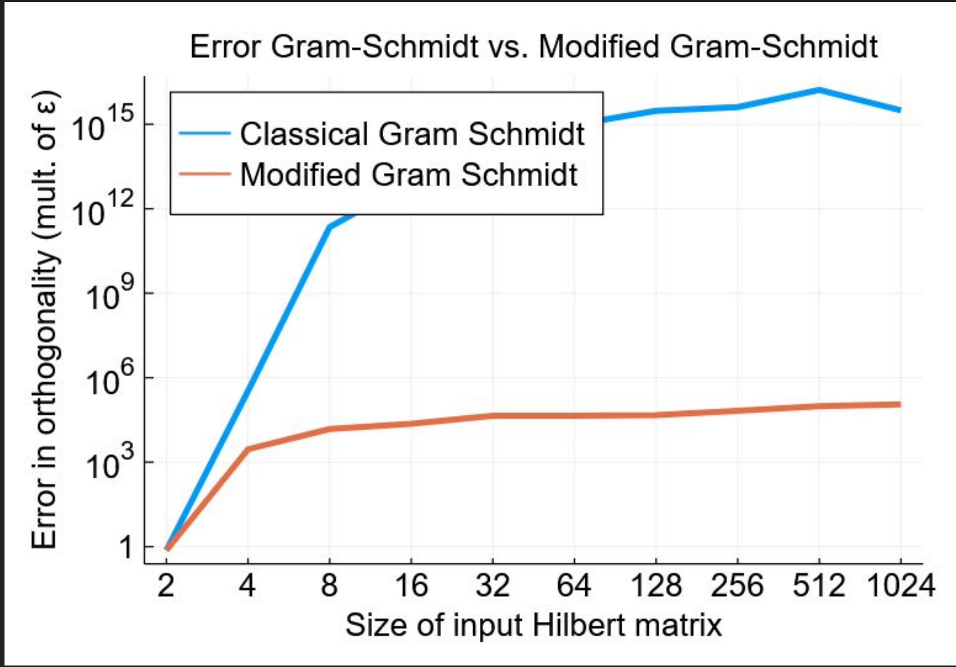


Figure reproduced from [Gram-Schmidt vs. Modified Gram-Schmidt | Laurent Hoeltgen]

Properties of QR Decomposition

$$Q^T Q = I$$

$$R = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 \\ 0 & 0 & 0 & 10 \end{bmatrix}$$

What is QR Decomposition used for?

- Linear least squares
- Basis for QR Eigenvalue algorithm
- Absolute value of a determinant of a square matrix
- Rank of matrix A

Design Challenge

- Limitations of sequential processors
- Efficient vector and matrix operations
- Fractional arithmetic without a dedicated floating point unit
- Mathematical routines without library support

Code Optimization Plan

- 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

- 24 functions
- 20 / 24 functions are 1 or more for loops
- Most functions perform some kind of simple vector operation e.g. element wise addition and subtraction, dot product, square root, and absolute value
- More complex functions require the use of multiple vector routines such as the Euclidean Norm which performs a dot product and square root

Base Code

```
void QR(SIZE_T rows, SIZE_T cols, DATA_T At[cols][rows], NUM_T Q[rows][rows], NUM_T R[rows][cols]) {
    assert(rows >= cols);
    NUM_T y[rows], q[rows];
    NUM_T y_norm;
    SIZE_T i, j;

    for (j=0; j<cols; ++j) {
        vec_copy(rows, At[j], y);

        for (i=0; i<j; ++i) {
            numt_copy_col(rows, cols, i, Q, q);
            R[i][j] = vec_dot(rows, q, y);
            vec_mulc(rows, q, R[i][j]);
            vec_sub(rows, q, y);
        }
        y_norm = l2_norm(rows, y);
        R[j][j] = y_norm;
        vec_divc(rows, y, y_norm);
        numt_set_col(rows, rows, j, y, Q);
    }
}
```

Base Code

```
NUM_T sqr_rt(NUM_T x, NUM_T eps, NUM_T tol, size_t max_iter) {
    assert((int)x >= 0);
    NUM_T x0 = (NUM_T) closest_perfect_square(x, MAX_ITER);
    NUM_T xn, err;
    NUM_T f_n, f_prime;
    size_t i;

    for (i=0; i<max_iter; ++i) {
        f_n = (x0 * x0) - x;
        f_prime = 2. * x0;

        if (abs_val(f_prime) < eps) {
            xn = 0.;
            break;
        }

        xn = x0 - (f_n / f_prime);
        err = abs_val(xn - x0);
        if (err <= tol) {
            break;
        }
        x0 = xn;
    }
    return xn;
}
```

For Loop Example

```c
void numt_copy_col(SIZE_T rows, SIZE_T cols, SIZE_T target_col, NUM_T src[rows][cols], NUM_T dest[rows]) {
        SIZE_T i;
        for (i=0; i<rows; ++i) {
                dest[i] = src[i][target_col];
        }
}
```

```c
NUM_T sqr_rt(NUM_T x, NUM_T eps, NUM_T tol, size_t max_iter) {
        assert((int)x >= 0);
        NUM_T x0 = (NUM_T) closest_perfect_square(x, MAX_ITER);
        NUM_T xn, err;
        NUM_T f_n, f_prime;
        size_t i;

        for (i=0; i<max_iter; ++i) {
                f_n = (x0 * x0) - x;
                f_prime = 2. * x0;

                if (abs_val(f_prime) < eps) {
                        xn = 0.;
                        break;
                }

                xn = x0 - (f_n / f_prime);
                err = abs_val(xn - x0);
                if (err <= tol) {
                        break;
                }
                x0 = xn;
        }
        return xn;
}
```

```c
int closest_perfect_square(DATA_T x, size_t max_iter) {
        int sq = 0, xn = 1;
        size_t i;

        for (i=0; i<max_iter; ++i) {
                sq = xn * xn;
                if (sq > (int)x) {
                        break;
                }
                xn += 1;
        }
        return xn;
}
```

Optimization 1: Loop Unrolling

```
void vec_sub(SIZE_T size, NUM_T v1[size], NUM_T v2[size]) {
        SIZE_T i;

        for (i=0; i<size; ++i) {
                v2[i] -= v1[i];
        }
}
```

```
//loop unrolling done
void vec_sub(SIZE_T size, NUM_T v1[size], NUM_T v2[size]) {
        SIZE_T i;

        for(i=0; i<size; i+=2){
                v2[i] -= v1[i];

                if(i+1 != size){
                        v2[i+1] -= v1[i+1];
                }
        }
}
```

```c
//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];
                }
    }
}
```

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

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

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

Optimization 2: Cache Oblivious Matrix Transpose

```
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];
                }
        }
}
```

```c
//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);
        }
}
```

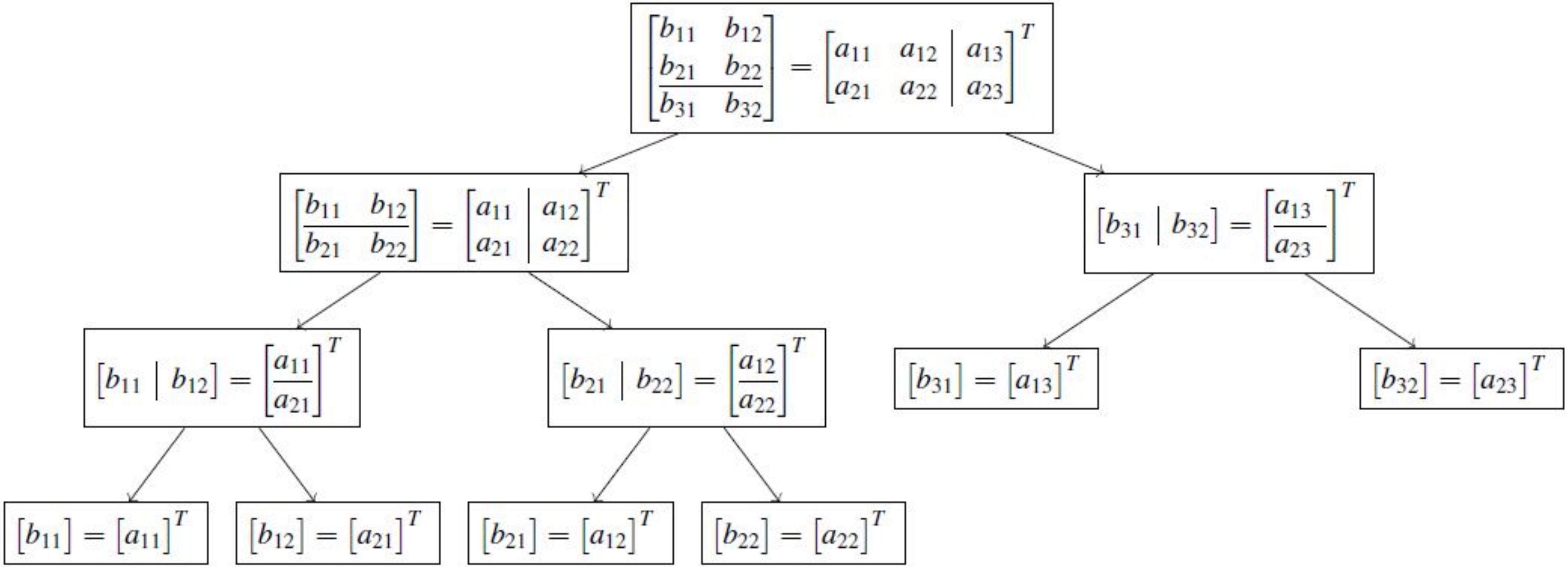
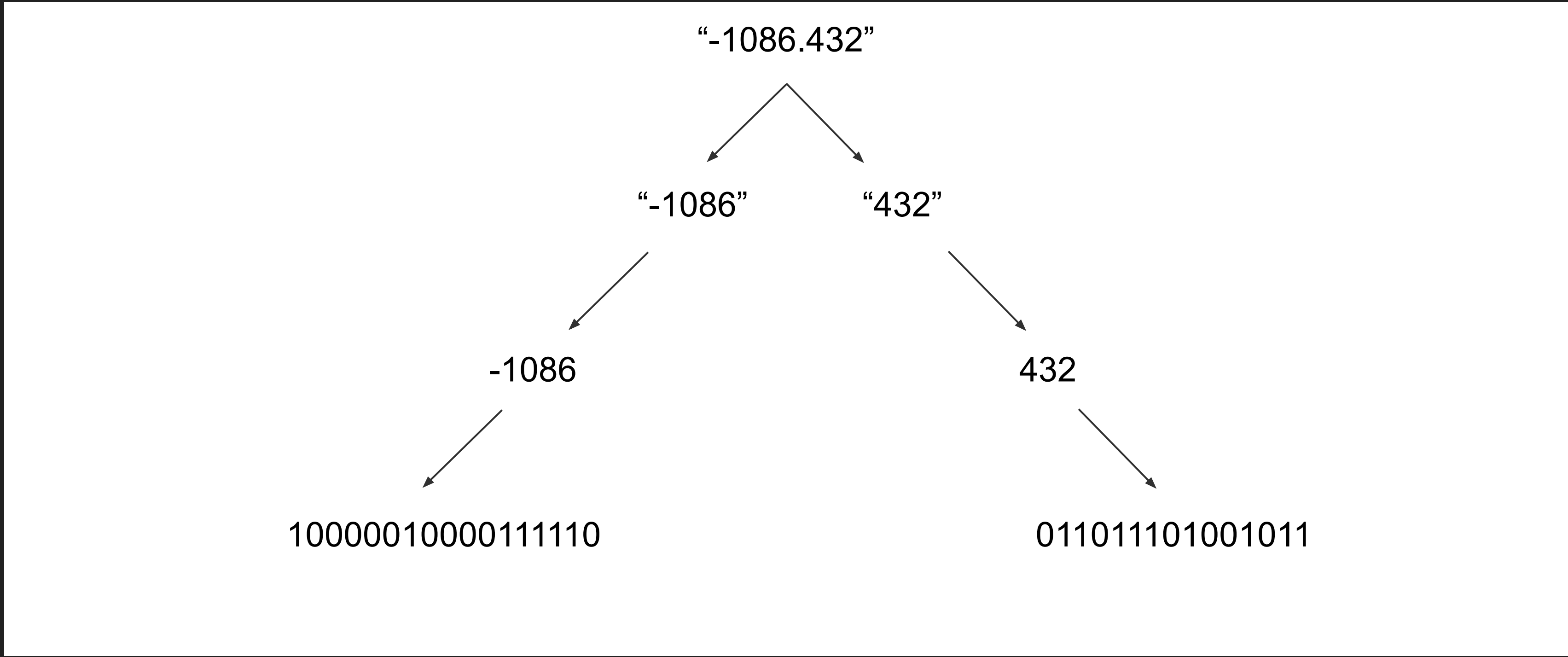


Image from SENG 475 Lecture Slides by Michael Adams

Optimization 3 - Fixed Point Arithmetic



Fixed Point Arithmetic - String to Fixed Width Integer

```
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;
}
```

Fixed Point Arithmetic - Fixed Width Integer to String

```
void bin_fx_to_str(char *s, UFX_T x) {
    FX_SIZE_T max_chars = FX_MAX_BIN_CHARS - 1;
    UFX_T stack = 0;
    FX_SIZE_T i = 0;

    for (; i < FX_SIZE; ++i) {
        stack <<= 1;
        stack |= (x & 1);
        x >>= 1;
    }

    i = 0;

    if (stack & 1) {
        s[0] = '-';
        i = 1;
        ++max_chars;
    }

    s[FX_WHOLE_BITS + i + 1] = '.';

    for (; i < max_chars; ++i) {
        if ('.' == s[i]) continue;
        s[i] = '0' + (stack & 1);
        stack >>= 1;
    }
    s[FX_MAX_BIN_CHARS] = '\0';
}
```

Fixed Point Arithmetic - Decimal to Binary fractions

```
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;
}
```

Assembly Comparison - vec_sub Base Version

```
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
```

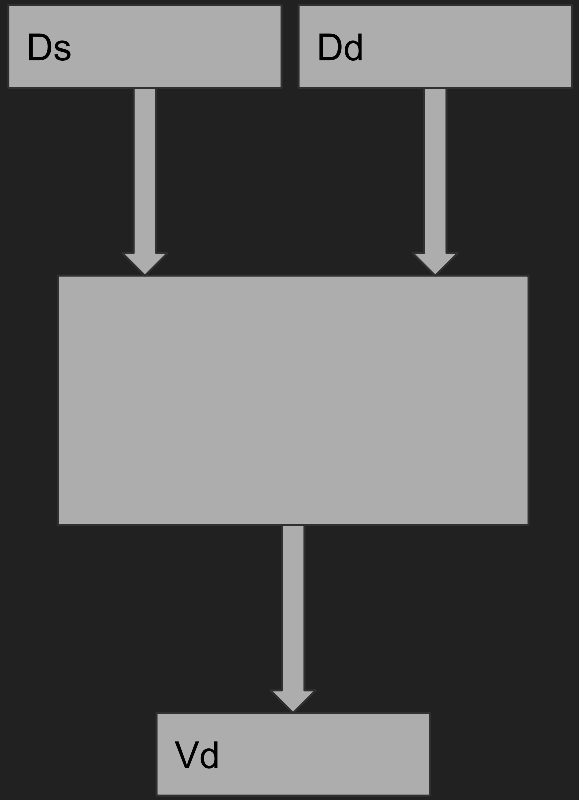
```
.L30:
        add     r1, r7, r4
        ldmia   r1, {r2-r3}
        ldmia   r5, {r0-r1}
        bl      __aeabi_dsub
        add     r4, r4, #8
        cmp     r4, r6
        stmia   r5!, {r0-r1}
        bne     .L30
.L31:
        ldmfd   sp!, {r4, r5, r6, r7, r8, lr}
        bx      lr
        .size   vec_sub, .-vec_sub
        .align  2
        .global numt_vec_copy
        .type   numt_vec_copy, %function
```

Assembly Comparison - vec_sub Loop Unrolled Version

```
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
```

Dedicated square root instruction



Compile and Run Commands

```
arm-linux-gcc -static -O2 qr.c -o qr.exe
```

```
./qr.exe m20x20.txt out.txt 20 20
```

Numerical Results - Run 100 Times for Average

Base version runtime: 0.739600 seconds

Loop unrolling added: 0.736900 seconds

For loop optimization added: 0.736600 seconds

Cache-oblivious matrix transpose added: 0.736400 seconds

Macros added: 0.736600 seconds

Operator strength reduction added: 0.738400 seconds

Unnecessary loop branches removed: 0.745700 seconds

Matrix transpose run once with 20x20 matrix

```
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%   )
```

Conclusion

- QR decomposition is challenging to optimize when there are many function calls or loops required
- Cache-oblivious matrix transpose lowers cache misses but does not lower runtime
- -O2 provides significant optimization
- FPA is challenging to implement

Future Work

- MLA instruction to parallelize vector and matrix routines
- Implement more robust rounding schemes for decimal to fixed point conversion to reduce round-off error
- Integrate existing QR decomposition code with custom Fixed Point library
- More robust testing procedures which measure round-off and truncation error, and runtime for matrices of arbitrary size and conditioning
- Research other QR decomposition algorithms