Matrix Diagonalization

SENG440: Embedded Systems

By: Michail Roesli & Robbie Tulip

Contents

Introduction:
Matrix Diagonalization

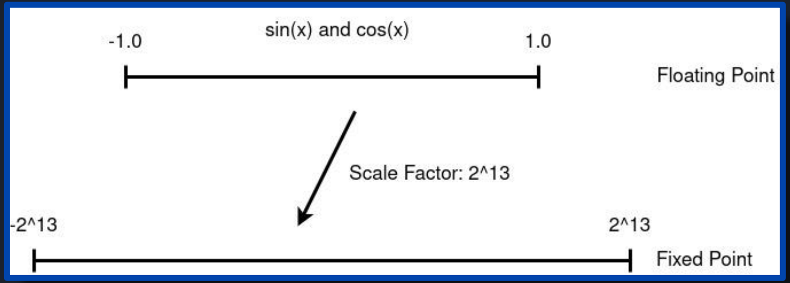
- Diagonalized matrices have desirable properties
- Matrix diagonalization is a process to diagonalize a matrix
  - Difficult to calculate quickly
- Used in various fields:
  - Wireless communications
  - Control applications
  - Image processing/compression
  - Molecular dynamics
  - Small angle scattering
  - Information retrieval
  - Etc.
- Singular Value Decomposition (SVD)
  - Complex algorithm to diagonalize a matrix

Introduction:
Problem Specification

- 4x4 square matrix input
- Real input values range $[-2^7, 2^7]$
- Use of 14 bit signed integers for fixed point arithmetic
- Output is returned as real values
- 32-bit ARMV7L processor

Background

- Jacobi Method
  - Iterative process for performing SVD
  - Moves energy onto the diagonal through rotation
  - Convergence



- Fixed-Point Arithmetic
  - Uses Scale factors
  - Fixed point notation - values that come before/after decimal point
  - State of mind
  - Addition increases word length by 1
  - Multiplication increases word length by 2x (2x-1 if unsigned val)
  - Division increases / decreases wordlength by the difference in bits between num and denom

Naive Implementation

- First task was to produce a naive unoptimised implementation
- Areas to optimize
  - Remove floating point operations
  - Arctangent/Sine/Cosine operations
  - Matrix multiplication
  - Remove unnecessary copies

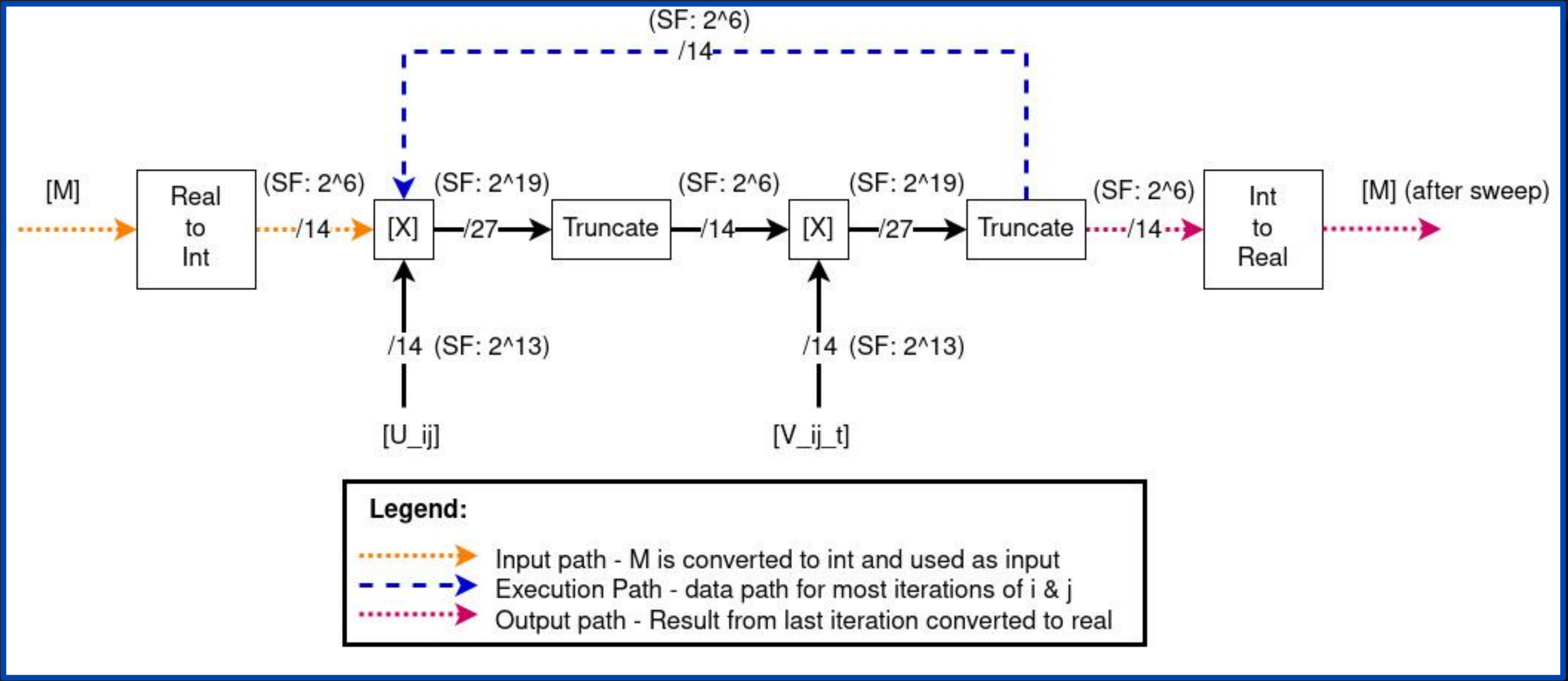
```
jacobi_sweep(matrix M, matrix U, matrix V){
    matrix M_prime, U_prime, V_prime;
    for (int i = 0; i < MATRIX_SIZE - 1; i++){
        for (int j = i; j < MATRIX_SIZE; j++){
            // Calculate sin(theta_l), cos(theta_l).
            // Init U_ij, U_ij_trans, and V_ij_trans
            // U * U_ij_trans -> U`
            // U_ij * M -> M`tmp
            // M`tmp * V_ij_trans -> M`
            // V_ij_trans * V_trans -> V_trans`
            // Copy M`, U`, and V_trans` into M, U, and V_trans
        }
    }
}
```

Optimization 1: Reducing copies.

- Change to algorithm
- Use two matrices for calculations:
  - Input
  - Output
- Swap matrices every iteration
- Reduces 96 copies to 16

```
jacobi_sweep(matrix M, matrix U, matrix V)
{
    matrix M_prime_1, U_prime_1, V_prime_1;
    matrix M_prime_2, U_prime_2, V_prime_2;
    matrix *M_input = &M_prime_1, M_output = &M_prime_2;
    matrix *U_input = &U_prime_1, U_output = &M_prime_2;
    matrix *V_input = &V_prime_1, V_output = &V_prime_2;
    for (int i = 0; i < MATRIX_SIZE - 1; i++)
    {
        for (int j = i; j < MATRIX_SIZE; j++)
        {
            // Calculate sin(theta_l), cos(theta_l).
            // Init U_ij, U_ij_trans, and V_ij_trans
            // U * U_ij_trans -> U`
            // U_ij * M -> M`tmp
            // M`tmp * V_ij_trans -> M`
            // V_ij_trans * V_trans -> V_trans`
            // Swap input and output pointers
        }
    }
    // Copy input matricies into M, U, and V.
}
```

Optimization 2: Fixed-Point Arithmetic



Optimization 3: Trigonometric Functions

- Native trigonometric functions use floating point operations
  - Mult, div, powers - Costly
  - Need to convert to fixed-point notation - more computation
- Multiple methods of optimization
  - Piecewise linear approximation
  - Lookup tables
- Chose Lookup Tables
  - More customizable
- Lookup Table Cost
  - Convert fixed point integer to table index
  - Requires a division, a shift, and some multiplication
  - Less accurate, more performant

Optimization 4: Matrix Multiplication

- Naive matrix multiplication is $O(n^3)$ operation

```
matrix_multiply(matrix LHS, matrix RHS, matrix RESULT){
    for (int i = 0; i < MATRIX_SIZE; i++){
        for (int j = 0; j < MATRIX_SIZE; j++){
            RESULT[i][j] = 0;
            for (int k = 0; k < MATRIX_SIZE; k++){
                RESULT[i][j] += LHS[i][k] * RHS[k][j];
            }
        }
    }
}
```

Optimization 4:
Matrix Multiplication - Continued

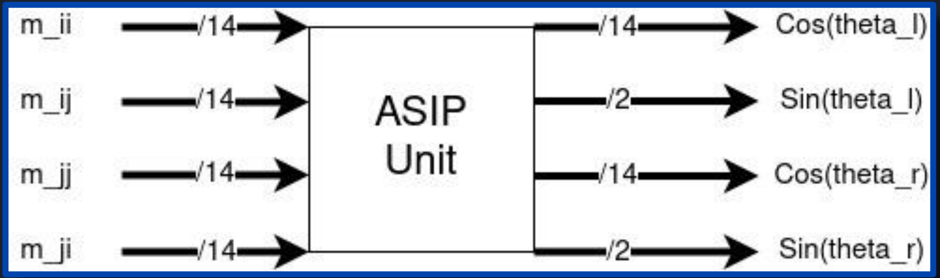
- Matrix Multiplication is O(n) with SIMD vector operations

```
matrix_multiply(matrix LHS, matrix RHS, matrix RESULT){
    int32x4_t RHS_row_0 = load_int32x4(RHS[0][0]);
    int32x4_t RHS_row_1 = load_int32x4(RHS[1][0]);
    int32x4_t RHS_row_2 = load_int32x4(RHS[2][0]);
    int32x4_t RHS_row_3 = load_int32x4(RHS[3][0]);
    int32x4_t RESULT_row;

    for (int i = 0; i < MATRIX_SIZE; i++){
        RESULT_row = multiply_int32x4_by_scalar(RHS_row_0, LHS[i][0]);
        RESULT_row += multiply_int32x4_by_scalar(RHS_row_1, LHS[i][1]);
        RESULT_row += multiply_int32x4_by_scalar(RHS_row_2, LHS[i][2]);
        RESULT_row += multiply_int32x4_by_scalar(RHS_row_3, LHS[i][3]);
        RESULT[i][0] = store_int32x4(RESULT_row);
    }
}
```

Theoretical Improvements: Application Specific Hardware

- Considered the benefits of a theoretical hardware instruction
- Instruction input:
  - 4x 14 bit integers packed into two 32 bit integers
- Instruction output:
  - 1x 32 bit output
    - 2x 14 bit output
    - 2x 2 bit output
- Expected Benefit:
  - Without ASIP: ~1500 clock cycles
  - Assume ASIP takes ~1000 clock cycles
  - Expect 33% speed up



Future Work

- Input variation: complex values and matrix size
- Lookup table
  - Input accuracy improvement - rounding
  - Cache misses - hit rate
  - Experiment with table parameters - size, density
    - Size vs accuracy tradeoffs
- Choosing next 2x2 matrix rotation optimally
- Code review and optimization
  - Remove branching where possible
- Problems
  - Energies decreasing after convergence

Conclusion

- Matrix diagonalization program in C
- 32-bit ARM machine
- Jacobi Method
- Fixed point arithmetic
- Lookup Table
- Parallelization
- Speed-Accuracy trade-off