UNIVERSITY OF VICTORIA

Department of Software Engineering
SENG 440 – Embedded Systems

Embedded Systems: Project Progress Report

June 22, 2020

Michail Roesli - V00853253 - mroesli@uvic.ca
Robert Tulip - V00846133 - rtulip@uvic.ca

Project

The project we will be working on is matrix diagonalization using Singular Value Decomposition (SVD). This algorithm has applications in several domains including wireless communications and control. Given a complex valued matrix, SVD attempts to decompose the matrix into two orthogonal / unitary matrices while the third is the diagonalized matrix. In this project we'll explore how to do this using the Jacobi method which reduces the off-diagonal elements to zero by applying rotations to the original matrix to eventually transform into a diagonalized matrix.

Problem

In this section we'll describe the main aspects to the SVD problem and what are the requirements and constraints.

Description

In the Cyclic Jacobi method we start by providing a complex value matrix in the following form.

$$M = U\Sigma V^T$$

Where Σ is a diagonal matrix of singular values, U & V are orthogonal / unitary matrices, and M is the real / complex valued matrix. To decompose the matrix, you must iterate over the matrix over several iterations using a cyclic order. In each iteration, 2x2 rotations are applied where the off diagonal elements are forced to zero. After several sweeps, we can obtain the final diagonalized matrix result.

To accomplish the rotations, we must calculate the rotation angles and then perform the rotations on the 2x2 matrices. To obtain the rotation angles, we compute the inverse tangents using the elements in the 2x2 matrices and then separate the theta l and theta r using the sum and diff equations.

$$\theta_{SUM} = \theta_r + \theta_l = arctan(\frac{m_{ji}+m_{ij}}{m_{jj}-m_{ii}}) \text{ and } \theta_{DIFF} = \theta_r - \theta_l = arctan(\frac{m_{ji}-m_{ij}}{m_{jj}+m_{ii}})$$

Next we calculate the sin and cosine of these angles to obtain 2x2 matrices which are then multiplied to produce the 2x2 diagonalized sub matrix. After several sweeps we should eventually converge to the desired result. One important aspect to note, is that because we do not require to completely vanish the off diagonal elements, we can take most of the energy and move it to the main diagonal since most of the energy will be removed in the next iteration. That means we can afford cheaper and less accurate ways of implementing arctan cos and sin.

Requirements, constraints and assumptions

The main requirements for this project is that we implement the Jacobi method in C that takes a square matrix of 12 bit integers (12-bit word length), and must be able to execute on a 32 bit ARM machine. We will be assuming that we're only dealing with a real-valued input matrix and that it will be even in height and width in order to apply the 2x2 matrix rotation operations for simplicity.

Plan

To do this project, we broke down the implementation into 5 main parts.

1. The base algorithm without any optimizations
2. Convert floats to 12 bit signed integer representation
3. Fixed-point arithmetic for all the necessary operations
4. Trigonometric approximations / Lookup tables
    a. Consider piecewise linear approximations for arctan, sin, and cos
    b. Determine the maximum error for the three middle points
5. Explore more fine tuned optimizations
    a. Pipelining
    b. MAC - multiply and accumulate operation
    c. Introduce parallelization - SIMD operations

Once we complete all of these, we will further explore the theoretical improvement, and how the number of clock cycles could be improved if we were to implement an instruction for matrix rotations.

Progress

We have currently been able to fully implement the functionality for the base unoptimized solution as seen in Appendix A. Using this solution we have been able to obtain the results as shown in the example computation in the slides. We also met up with Mihai Sima to talk about the problem we had with the transpose in the first equation, how we plan on implementing the 2x2 diagonalization operation, and how to go about doing the fixed-point arithmetic trigonometric functions. We also implemented a template arctan function using the taylor series expansion about a point. In our implementation we have also made minimal code optimizations, such as minimizing cache misses in the for loops by looping by columns then rows.

Future Work

In the following phase we plan to implement the next sections we've outlined in our plan. This includes converting our double variables to 12 bit signed integers, and then using fixed-point arithmetic for all our operations. Once that is complete we'll explore more fine tuned optimizations which include:

- Software pipelining
- Precalculating the arctan of angles, such as 45 degrees
- Lookup tables for trigonometric functions
- Tchebishev polynomial for arctan approximation for interval
- Piecewise linear approximation with 3 middle points
- MAC
- Introduce parallelization with SIMD operations

We also plan to meet with Mihai Sima to discuss our parallelization plans and ensure that we are on the right track.

Additionally, time permitting, we'll try varying our accuracy to see potential performance improvements. Some other considerations we have is to explore input matrices with various properties such as symmetry and see how we can optimize it for those particular cases. Furthermore, we will consider the hypothetical speed up of various hardware implementations of complex functions currently implemented in software, such as the trigonometric functions or a 2x2 matrix diagonalization.

Appendix A: SVD C Implementation

main.c

```c
/*
 *
 * main.c
 *
 */

#include "config.h"
#include "svd.h"

/*  Define M, U, & V as globals for now.*/
double m[4][4] = {
    {31, 77, -11, 26},
    {-42, 14, 79, -53},
    {-68, -10, 45, 90},
    {34, 16, 38, -19},
};

double u[4][4] = {
    {1, 0, 0, 0},
    {0, 1, 0, 0},
    {0, 0, 1, 0},
    {0, 0, 0, 1},
};

double v_trans[4][4] = {
    {1, 0, 0, 0},
    {0, 1, 0, 0},
    {0, 0, 1, 0},
    {0, 0, 0, 1},
};

/**
```

```
 * @brief Helper function to print a 4x4 matrix
 *
 * @param mat
 */
void mat_print(double mat[4][4])
{
    for (int row = 0; row < 4; row++)
    {
        printf("[ ");
        for (int col = 0; col < 3; col++)
        {
            printf("%f, ", mat[row][col]);
        }
        printf("%f ]\n", mat[row][3]);
    }
}

int main(void)
{
    for (int i = 0; i < 4; i++)  // Perform a "sweep" 4 times.
    {
        sweep(m, u, v_trans);
        mat_print(m);
        printf("\n");
    }
}
```

svd.c

```
/*
 *
 * svd.c
 *
 */
```

```
#include "svd.h"

/**
 * @brief Helper function to multiply a (size x size) matrix
 *
 * Result is placed in out[][]
 *
 * @param size - The size of the two square matrices
 * @param LHS  - The left matrix
 * @param RHS  - The right matrix
 * @param out  - The output resulting matrix
 *
 * Modified version of algorithm from:
https://www.geeksforgeeks.org/c-program-multiply-two-matrices/
 */
void mat_mul(
    int size,
    double LHS[size][size],
    double RHS[size][size],
    double out[size][size])
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            out[i][j] = 0;
            for (int k = 0; k < size; k++)
            {
                out[i][j] += LHS[i][k] * RHS[k][j];
            }
        }
    }
}

/**
 * @brief Function to obtain arctan using integer / fixed-point
arithmetic
 * and taylor series expansion
 *
 * Result is placed in out[][]
 *
 * @param approximation
 * @param out
 * @return out
 */
double arctan(int approximation, double out)
{
    for (int i = 0; i < approximation; i++)
    {
        out += pow(-1, (i + 1)) * (1 / (i * 2 + 1)) * pow(out, (i * 2
+ 1));
    }
    return out;
}

/**
 * @brief Performs a single sweep of the svd algorithm
 *
 */
void sweep(double m[4][4], double u[4][4], double v_trans[4][4])
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = i + 1; j < 4; j++)
        {
            /**
             * Do all of the angle calculations
             *
             * TODO: Implement all of these functions.
             *      - atan
             *      - sin
             *      - cos
             *
             */
            double theta_sum = atan((m[j][i] + m[i][j]) / (m[j][j] -
m[i][i]));
```

```
            double theta_diff = atan((m[j][i] - m[i][j]) / (m[j][j] +
m[i][i]));
            double theta_l = (theta_sum - theta_diff) / 2;
            double theta_r = theta_sum - theta_l;
            double sin_theta_l = sin(theta_l);
            double cos_theta_l = cos(theta_l);
            double sin_theta_r = sin(theta_r);
            double cos_theta_r = cos(theta_r);

            /**
             * @brief Create temporary matrices for u_ij, u_ij_trans
and v_ij_trans
             *
             */
            double u_ij[4][4] = {
                {1, 0, 0, 0},
                {0, 1, 0, 0},
                {0, 0, 1, 0},
                {0, 0, 0, 1},
            };

            double u_ij_trans[4][4] = {
                {1, 0, 0, 0},
                {0, 1, 0, 0},
                {0, 0, 1, 0},
                {0, 0, 0, 1},
            };

            double v_ij_trans[4][4] = {
                {1, 0, 0, 0},
                {0, 1, 0, 0},
                {0, 0, 1, 0},
                {0, 0, 0, 1},
            };

            /**
             * U_ij = [ cos(θl) -sin(θl) ]
             *        [ sin(θl)  cos(θl) ]
```

```
 */
u_ij[i][i] = cos_theta_l;
u_ij[j][j] = cos_theta_l;
u_ij[i][j] = -sin_theta_l;
u_ij[j][i] = sin_theta_l;

/**
 * U_ij_Trans = [  cos(θl) sin(θl) ]
 *              [ -sin(θl) cos(θl) ]
 */
u_ij_trans[i][i] = cos_theta_l;
u_ij_trans[j][j] = cos_theta_l;
u_ij_trans[i][j] = sin_theta_l;
u_ij_trans[j][i] = -sin_theta_l;

/**
 * V_ij_Trans = [  cos(θr) sin(θr) ]
 *              [ -sin(θr) cos(θr) ]
 */
v_ij_trans[i][i] = cos_theta_r;
v_ij_trans[j][j] = cos_theta_r;
v_ij_trans[i][j] = sin_theta_r;
v_ij_trans[j][i] = -sin_theta_r;

/**
 * Create temporary matrices for calculations output.
 */
double u_prime[4][4];
double v_trans_prime[4][4];
double m_prime_tmp[4][4];
double m_prime[4][4];
// Do the calculations
// [U][U_ij_T] = [U']
mat_mul(4, u, u_ij_trans, u_prime);
// [U_ij][M] = [M'_tmp]
mat_mul(4, u_ij, m, m_prime_tmp);
// [M_tmp][V_ij_T] = [M']
mat_mul(4, m_prime_tmp, v_ij_trans, m_prime);
```

```
                // [V_ij][V_T] = [V'_T]
                mat_mul(4, v_ij_trans, v_trans, v_trans_prime);

                /**
                 * Copy the values into U, V, and M.
                 * TODO: Don't copy every iteration.
                 */
                for (int row = 0; row < 4; row++)
                {
                    for (int col = 0; col < 4; col++)
                    {
                        u[row][col] = u_prime[row][col];
                        v_trans[row][col] = v_trans_prime[row][col];
                        m[row][col] = m_prime[row][col];
                    }
                }
            }
        }
    }
```

svd.h

```
/*
 *
 * svd.h
 *
 */
#ifndef svd_h
#define svd_h

#include <math.h>

/**
```

```
 * @brief Performs a single sweep of the svd algorithm
 *
 */
void sweep(double m[4][4], double u[4][4], double v_trans[4][4]);

#endif
```

config.h

```
/*
 *
 * config.h
 *
 */

#ifndef CONFIG_H_
#define CONFIG_H_

#ifndef VERBOSE
#define VERBOSE 0
#endif

/* Includes */
#include <stdio.h>
#include <stdlib.h>

/* Main Definitions */
#endif /* CONFIG_H_ */
```