Matrix Inversion Optimization

Angus Morrison, V00937482, Software Engineering
Daisy Kwan, V00919518, Computer Science

Department of CS/ECE, University of Victoria
SENG440 Embedded Systems
angusmorrison@uvic.ca
daisykwan@uvic.ca

Constraints and Specifications

Specs:

- 10x10 matrix
- 12 bit width elements
- Use fix point arithmetic
- Use real value matrices
- *Condition Number threshold* < *9*

We want to:

- Vectorize the code

Processor:

- Arm
- Qemu emulator on SENG440 sever
- Supports SIMD (NEON) intrinsics

History

- Carl Friedrich Gauss developed Gaussian elimination
- Gauss-Jordan elimination initially described by B.-I. Clasen 1888
- Wilhelm Jordan expanded on Gaussian elimination in the same year

Background

- Gauss Jordan elimination
  - The use of three elementary row operations to transform a matrix into ***reduced row-echelon form***
  - A matrix is in reduced row-echelon form when all four conditions are met

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix} \sim \ldots \sim \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Possible Application of Interest

- Solving systems of equations

- Communications
  - 12 bits is commonly used in communication

- Computer graphics
  - Used in 3D renderings, ray casting, transformations, and physical simulations

Converting From Float to Integer

Bit Width: 12

- 1 bit sign
- 11 bits magnitude

$$-1.0 \quad \dots \quad +1.0$$
$$-2^{11} \quad \dots \quad +2^{11}$$

Scale Factor: 2^{11}

$$\begin{bmatrix} 0.99 & 0.92 & -0.86 \\ -0.56 & 0.32 & 0.53 \\ -0.63 & -0.88 & 0.29 \end{bmatrix} \longrightarrow$$

$$\begin{bmatrix} 0.99 \times 2^{11} & 0.92 \times 2^{11} & -0.86 \times 2^{11} \\ -0.56 \times 2^{11} & 0.32 \times 2^{11} & 0.53 \times 2^{11} \\ -0.63 \times 2^{11} & -0.88 \times 2^{11} & 0.29 \times 2^{11} \end{bmatrix} \longrightarrow$$

$$\begin{bmatrix} 2028 & 1884 & -1761 \\ -1147 & 655 & 1085 \\ -1290 & -1802 & 593 \end{bmatrix}$$

Steps taken 1 - Hard coding the identity matrix

Original Code

```
int main(int argc, char *argv[]) {
    float matrix[10][20] = {
        {10, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 9, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 8, 1, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 7, 1, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 6, 1, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 5, 1, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 4, 1, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 3, 1, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 2, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
    };

    printf("Starting the inversion\n");
    // Do the inversion
    /* Augmenting Identity Matrix of Order 10 */
    for (int i = 0; i < 10; i++) {
        for (int j = 0; j < 10; j++) {
            // if i == j, we are on the diagonal, so make the identity matrix
            if (i == j) {
                matrix[i][j + 10] = 1;
            } else {
                matrix[i][j + 10] = 0;
            }
        }
    }
```

Gauss-Jordan Code

```
int main(int argc, char *argv[]) {
    clock_t start_time;
    start_time = clock();

    float input_matrix[10][20] = {
        {1.0, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        {0.73, 0.9, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        {0.73, 0.73, 0.8, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        {0.73, 0.73, 0.73, 0.7, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        {0.73, 0.73, 0.73, 0.73, 0.6, 0.73, 0.73, 0.73, 0.73, 0.73, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0},
        {0.73, 0.73, 0.73, 0.73, 0.73, 0.5, 0.73, 0.73, 0.73, 0.73, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0},
        {0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.4, 0.73, 0.73, 0.73, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0},
        {0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.3, 0.73, 0.73, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0},
        {0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.2, 0.73, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0},
        {0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.73, 0.1, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0},
    };
```

Steps taken 2 - Register keyword

Original

```
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
```

Optimized

```
register int i = 0;
register int j = 0;
register int k = 0;
```

Steps taken 2 - Register keyword (Assembly)

Original

```
.L13:
        ldr     r3, [fp, #-60]
        add     r3, r3, #1
        str     r3, [fp, #-60]
.L11:
        ldr     r3, [fp, #-60]
        cmp     r3, #9
        ble     .L14
        ldr     r3, [fp, #-64]
        add     r3, r3, #1
        str     r3, [fp, #-64]
```

Optimized

```
        ldr     r1, [fp, #-1896]
        sub     r1, r1, #1
        str     r1, [fp, #-1896]
.L23:
        ldr     r2, [fp, #-1896]
        cmp     r2, #0
        bge     .L24
        ldr     r3, [fp, #-1900]
        sub     r3, r3, #1
        str     r3, [fp, #-1900]
```

This shows in two places that using the register keyword didn’t do anything as the program still loads the value for the counters decrements and then stores the value back in memory

Steps taken 3 - Loop decrement

Original

```
// Devide each row by the diagnal to get the inverse
for (int i = 0; i < 10; i++) {
    float diagonal = matrix[i][i];
    for (int j = 0; j < 20; j++) {
        matrix[i][j] /= diagonal;
    }
}
```

Optimized

```
// Divide each row by the diagonal to get the inverse
// move diagnal to a register and so that it is not reinitialized each loop
register float diagonal;
for (i = 9; i >= 0; i--){
    diagonal = matrix[i][i];
    for (j = 19; j >= 0; j--) {
        matrix[i][j] /= diagonal;
    }
}
```

Steps taken 3 - Loop decrement (Assembly)

```
.L23:
      ldr   r2, [fp, #-1896]
      cmp   r2, #0
      bge   .L24
      ldr   r3, [fp, #-1900]
      sub   r3, r3, #1
      str   r3, [fp, #-1900]
.L22:
      ldr   r4, [fp, #-1900]
      cmp   r4, #0
      bge   .L25
```

```
.L25:
      mov   r1, #19
      str   r1, [fp, #-1896]
      b     .L23
```

This shows that the loop decrement didn’t change much in the code because the register keyword didn’t work. Due to the load and store we weren’t able to check the zero flag to speed up the code and instead we still have to compare to zero

Steps taken 4 - Loop removal

Original

```
printf("Starting the inversion\n");
// Do the inversion
/* Augmenting Identity Matrix of Order 10 */
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        // if i == j, we are on the diagonal, so make the identity matrix
        if (i == j) {
            matrix[i][j + 10] = 1;
        } else {
            matrix[i][j + 10] = 0;
        }
    }
}

// Interchange the rows of the matrix, starting from the last row
for (int i = 9; i > 0; i--) {
    // Swap rows
    if (matrix[i - 1][0] < matrix[i][0]) {
        float temp[20];
        for (int j = 0; j < 20; j++) {
            temp[j] = matrix[i][j];
            matrix[i][j] = matrix[i - 1][j];
            matrix[i - 1][j] = temp[j];
        }
    }
}
```

Optimized

```
float matrix[10][20] = {
    {10, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1 , 9, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {1 , 1, 8, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0},
    {1 , 1, 1, 7, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0},
    {1 , 1, 1, 1, 6, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0},
    {1 , 1, 1, 1, 1, 5, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0},
    {1 , 1, 1, 1, 1, 1, 4, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0},
    {1 , 1, 1, 1, 1, 1, 1, 3, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0},
    {1 , 1, 1, 1, 1, 1, 1, 1, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0},
    {1 , 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
};

register int i = 0;
register int j = 0;
register int k = 0;

// printf("Apply Gauss-Jordan elimination\n");
// Interchange the rows of the matrix, starting from the last row

// create local variable so that memory only has to be accessed once per loop
// I don't think we can use the register keyword as we would run out of registers to use
// I removed the nested loops and replaced them with memcpy
float array_temp[20];
for(i = 8; i>0; i--){
    if(matrix[i][0] < matrix[i+1][0]){
        memcpy(array_temp, matrix[i+1], 20*sizeof(int));
        memcpy(matrix[i+1], matrix[i], 20*sizeof(int));
        memcpy(matrix[i], array_temp, 20*sizeof(int));
    }
}
```

Steps taken 5 - Vectorization

Original

```
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        if (j != i) {
            float temp = matrix[j][i] / matrix[i][i];
            for (int k = 0; k < 20; k++) {
                matrix[j][k] -= matrix[i][k] * temp;
            }
        }
    }
}
```

Optimized

```
for (i = 9; i >= 0; i--) {
if(int_matrix[i][i] == 0){
    printf("dividing by 0 at i=%d\n", i);
    return 0;
}
    for (j = 9; j >=0; j--) {
        if (j != i) {
    temp = (int_matrix[j][i] * 2048)/ int_matrix[i][i]; // Have to ensure the scale factor remains
    for(k=0; k<20; k = k+4){ // runs 5 times
        // load the vector values
        i_vector_row = vld1q_u32(int_matrix[i] + k);
        j_vector_row = vld1q_u32(int_matrix[j] + k);
        // Do the multiplication
        i_vector_row = vmulq_n_u32(i_vector_row, temp);
        i_vector_row = vmulq_n_u32(i_vector_row, 1/2048); // have to ensure the scale factor remains
        // vector subtraction
        j_vector_row = vsubq_u32(j_vector_row, i_vector_row);
        // store the vector back into the proper space
        vst1q_u32(int_matrix[j] + k, j_vector_row);
    }
        }
    }
```

Using memcpy() - Memory accesses

```
for (int i = 9; i > 0; i--) {
    // Swap rows
    if (matrix[i - 1][0] < matrix[i][0]) {
        float temp[20];
        for (int j = 0; j < 20; j++) {
            temp[j] = matrix[i][j];
            matrix[i][j] = matrix[i - 1][j];
            matrix[i - 1][j] = temp[j];
        }
    }
}
```

```
int array_temp[20];
for(i = 8; i>0; i--){
    if(int_matrix[i][0] < int_matrix[i+1][0]){
        memcpy(array_temp, int_matrix[i+1], 20*sizeof(int));
        memcpy(int_matrix[i+1], int_matrix[i], 20*sizeof(int));
        memcpy(int_matrix[i], array_temp, 20*sizeof(int));
    }
}
```

Using memcpy didn’t reduce the number of memory access operations from the original to the optimized code however it did allow us to completely remove a loop freeing a register and reducing the number of operations

Results

- The optimization increased the number lines in the assembly file from 854 to 871
- This was completely different than what we were expecting the result to be
- There was no change in program runtime to the 1000th of a second
- There was no change in runtime with ill and well conditioned matrices

Results Unoptimized

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

              1.06 msec task-clock:u              #    0.133 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
               342      page-faults:u             #    0.322 M/sec
         1,422,046      cycles:u                  #    1.338 GHz
         1,891,280      instructions:u            #    1.33  insn per cycle
           368,907      branches:u                #  347.009 M/sec
            11,396      branch-misses:u           #    3.09% of all branches

       0.007993440 seconds time elapsed

       0.001003000 seconds user
       0.001003000 seconds sys
```

Results Optimized

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

              1.11 msec task-clock:u              #    0.146 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
               341      page-faults:u             #    0.308 M/sec
         1,407,980      cycles:u                  #    1.270 GHz
         1,890,799      instructions:u            #    1.34  insn per cycle
           368,818      branches:u                #  332.729 M/sec
            11,248      branch-misses:u           #    3.05% of all branches

       0.007566833 seconds time elapsed

       0.000571000 seconds user
       0.001714000 seconds sys
```

Conclusion

We do not recommend trying to optimize matrix inversion code manually. The amount of coding and debugging time that went in for an unnoticeable change in runtime does not justify the costs.

Additionally there was no noticeable change in runtime between a ill and well conditioned matrix. This leads us to believe that the system is more robust albeit slow.

References

Brilliant. (n.d.). *Gauss-Jordan Elimination*. Brilliant Math & Science Wiki. https://brilliant.org/wiki/gaussian-elimination/#:~:text=Carl%20Friedrich%20Gauss%20championed%20the,inverses%2C%20Gauss%2DJordan%20elimination.

Knill, O. (n.d.). *About Gauss-Jordan elimination*. Math 22a Harvard College Fall 2018. https://people.math.harvard.edu/~knill/teaching/math22a2018/exhibits/gaussjordan/index.html

M.7 Gauss-Jordan Elimination: Stat Online. PennState: Statistics Online Courses. (n.d.). https://online.stat.psu.edu/statprogram/reviews/matrix-algebra/gauss-jordan-elimination#:~:text=The%20purpose%20of%20Gauss%2DJordan,the%20bottom%20of%20the%20matrix