Matrix Inversion Using Gauss-Jordan Elimination

Department of CS/ECE, University of Victoria
SENG440 Embedded Systems

Angus Morrison
Software Engineering
V00937482
angusmorrison@uvic.ca

Daisy Kwan
Computer Science
V00919518
daisykwan@uvic.ca

August 9, 2023

Submission Date: _ _ _ _ August 9, 2023 _ _ _ _

Table of Contents

List of Figures

1 Introduction

1.1 Domain

Linear algebra is the branch of mathematics concerning linear equations, linear mappings, and how they are represented in vector spaces using matrices and vectors [1]. In linear algebra, the inverse of a matrix is used to solve a system of linear equations. Real-world applications of matrix inverse include Multiple-Input, Multiple-output (MIMO) wireless communication, computer graphics, and real-time renderings [2].

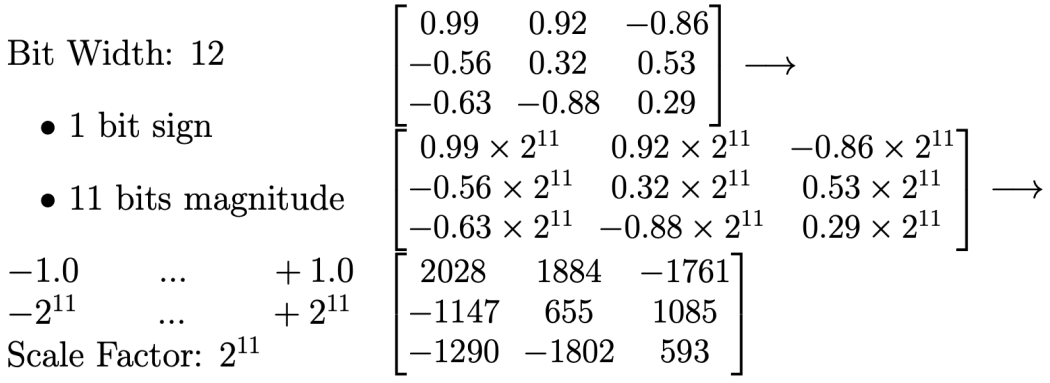
1.2 Purpose

The purpose of this project is to write a program for matrix inversion using Gauss-Jordan Elimination and apply optimization techniques to improve the performance of the program developed.

1.3 Constraints and Specifications

1.3.1 Project Specifications

For this project, the matrix inverse program is for a 10x10 matrix with 12-bit width elements. The matrix must be a real value matrix and only fixed point arithmetic may be used to convert matrix elements to real values. The condition number threshold for our program is a condition number less than 9.



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

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

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

Figure 1. Example of fixed point arithmetic to convert to integer

1.3.2 Processor

This project runs on the QEMU emulator on the SENG 440 server which emulates an ARM instruction set. This processor also supports SIMD (NEON) intrinsics. We will be using

the GCC compiler with the static command in the temp partition on the UGLS server hosted by UVic.

1.3.3 Performance Requirements

The optimization done on the program should aim to provide some speed-up. Furthermore, cache misses should be observed and minimized in the optimized version.

1.4 Contributions

This project consists of three main portions, the program code, project report, and presentation. The program code and optimization is mainly written by Angus Morrison with the help of online resources and discussion/collaboration with project partner, Daisy Kwan, during in-person project meetings. ChatGPT was also used in the debugging process of the initial unoptimized code. However, ChatGPT was not used in the optimization portion of the project. The project report is mostly written by Daisy Kwan with contributions from Angus Morrison. Lastly, the presentation is prepared equally by both Angus and Daisy.

1.5 Project Organization

To start the project, we first did basic research on the Gauss-Jordan elimination to get a good understanding of the necessary steps in the algorithm. Then we proceeded to write the unoptimized code for our program. With the help of online resources, such as online guides and ChatGPT, we were successful in coming up with a bug-free unoptimized matrix inversion program using Gauss-Jordan. From there, we started writing the optimized version of the program. As stated before, we did not use any guides or ChatGPT, but we did use Google in the optimization. To begin optimization, we started by making small changes to our code, such as loop decrement and using the register keyword. At the same time, we also started preparing our presentation slides for the in-class dry-run presentation. Based on the feedback from our professor, the main optimization we added was vectorizing the code.

2 Theoretical Background

2.1 Brief History and Process

Gauss-Jordan elimination is based on the row reduction process called Gaussian elimination named after Carl Friedrich Gauss in 1810, a German mathematician, physicist, and geodesist. Gauss' process uses elementary row operations to manipulate the matrix into as close to an upper-triangular matrix as possible. These elementary row operations consist of the following [3]:

1. Swapping two rows
2. Multiplying a row by a constant
3. Adding a multiple of one row to another

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

$$\mathrm{R2} = \mathrm{R2} + (\text{-}1)\mathrm{R1} \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & -2 \\ 2 & 2 & 2 \end{bmatrix}$$

$$R3 = R3 + (-2)R1 \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & -2 \\ 0 & -2 & -4 \end{bmatrix}$$

$$R2 = (-1)R2 \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & -2 & -4 \end{bmatrix}$$

$$R3 = R3 + 2R2 \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

Figure 2. Example of Gaussian Elimination

In Figure 1, from the first matrix, adding a multiple -1 of row 1 to row 2 gives the new row 2 as shown in the second matrix. Then, adding a multiple -2 of row 2 to row 3 makes the element under the first pivot in row 1 zero, shown in the third matrix. Next, multiply row 2 by a multiple of -1 as shown in the fourth matrix. Lastly, adding a multiple 2 of row 2 to row 3 makes the element under the pivot in row 2 zero, shown in the fifth matrix. Now, the matrix is in upper-triangular form.

This Gaussian elimination process gives us a matrix that is in *row echelon form*. Wilhelm Jordan later expanded on the Gaussian elimination method in 1888 to further reduce the matrix into *reduced row echelon form*. This is now known as the Gauss-Jordan elimination [3]. For a matrix to be in reduced row echelon form, the following must be satisfied [4]:

1. All rows containing zeros are at the bottom of the matrix
2. The leading entry (non-zero value, also known as a pivot) of a row is to the right of the leading entry of the row above
3. Leading entries must be 1
4. All values in a column containing a leading entry 1 are 0

$$\begin{bmatrix} 1 & 2 & 3 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix}$$

$$\mathrm{R2} = \mathrm{R2} + (\text{-}1)\mathrm{R1} \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & -2 \\ 2 & 2 & 2 \end{bmatrix}$$

$$R3 = R3 + (-2)R1 \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & -2 \\ 0 & -2 & -4 \end{bmatrix}$$

$$R2 = (-1)R2 \rightarrow \begin{bmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \\ 0 & -2 & -4 \end{bmatrix}$$

$$R1 = R1 + (-2)R2 \rightarrow \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 2 \\ 0 & -2 & -4 \end{bmatrix}$$

$$R3 = R3 + 2R2 \rightarrow \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

Figure 3. Example of Gauss-Jordan Elimination

In Figure 2, elementary row operations applied to Figure 1 are also applied. But in addition to that, an extra step, adding a multiple -2 of row 2 to row 1 also eliminates the element above the pivot in the second row, which satisfies property 4, “All values in a column containing a leading entry 1 is 0”. Lastly, take note that the row of zeros is at the bottom of the matrix.

$$\left[\begin{array}{ccc|ccc} 1 & 6 & 2 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 3 & 1 & 0 & 0 & 1 \end{array}\right] \rightarrow \left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 1 & 0 & -2 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & -3 & 1 \end{array}\right]$$

Figure 4. Example of an augmented matrix used to calculate the matrix inverse

2.2 ARM Processor and Instruction Set

As we are using the QEMU ARM system emulator on the SENG 440 server, we have the option to emulate a 32-bit or 64-bit ARM CPU. For this project, we emulated a 32-bit ARM

machine. ARM processors are based on reduced instruction set computer (RISC) architecture [5]. RISC uses very simple instructions that perform only one function, meaning each instruction can be executed quickly. Though this would mean possibly more code is required to complete a task. The simpler instructions also make it easier to have an instruction pipeline which allows for instruction parallelism which would then increase the performance speed [6][7]. As we are using a 32-bit ARM processor for this project, we followed the 32-bit ARM instruction set.

3 Design Process

3.1 Optimization

3.1.1 Register Keyword

By using the registered keyword, we were able to guide the compiler as to which variables we wanted to be held in registers instead of in memory. As shown in the assembly code below this was not addressed correctly by the compiler.

```
.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
```

Figure 5. Assembly Code Snippet of Register Keyword

The variables that we asked to be held in registers were `i`, `j`, `k`, `input_norm`, `output_norm`, and `temp`. These variables were used for loop counters and operations inside the loops. Had the compiler recognised and accepted the register keyword many of the load and store operations would have been removed from the assembly code.

3.1.2 Decrement Loops

In the unoptimized code, we used incrementing for loops to loop through the matrix elements, however, in the optimized code, we chose to mostly use decrement loops. We made this choice because the comparison to 0 costs a total of two cycles, one cycle for the

decrementation of the loop counter, and the comparison to 0 itself costs one cycle. Compared to an incrementing loop which costs a total of three cycles, one cycle for loop counter incrementation, one cycle to subtract constant N, and one more cycle to check the result.

While it was expected that loop decrement would reduce the time the program took, because that the register keyword was not addressed correctly by the compiler, this decrement of loops did not affect the comparison. Due to the counter variable being stored in memory, it needed to be loaded compared and stored every time the loop concluded a cycle.

3.1.3 Code Vectorization

One of the optimization methods that we implemented was code vectorization. By using the NEON intrinsic included in ARM, we were able to perform vector operations on part of each row of the matrix. Due to the matrix size being 10x10 and the NEON intrinsic being limited to using vectors of length 4 at most, we had to loop through the vector operations five times to operate on the entire matrix. The reason we had to loop through five times, is because we appended the identity matrix to the end of our matrix creating a 10x20 matrix.

By using NEON, we were able to access another set of registers. These registers are dedicated to vector operations and, as such, are not used by the base program. By having access to these registers, we were able to move more data out of memory at one time and reduce memory operations. We expected vectorizing the code to have a major impact on the performance of the program. However, after testing, we found that this did not improve the speed of the program as much as expected.

3.1.4 Using `memcpy()` and Loop Removal

In the optimized_code.c file, we used `memcpy()`, rather than looping through each element in the row of the matrix as seen in the unoptimized_code.c file when doing the row swapping. We choose to do this because we wanted to once again reduce the number of memory accesses. In our code, we `used memcpy()` to copy the entire row in the matrix, and to do the swapping. Using `memcpy()` allowed us to streamline the code, and allowed us to remove the nested for loop. We were not able to determine the cache misses because `memcpy()` is dependent on the compiler and the version of C.

3.2 Other Designs Changes

In the optimized_code.c file lines 12 to 23 we also reorganized our data by hard coding the identity matrix with the matrix we tested with. This allows us to remove the code to augment

the matrix seen in the unoptimized_code.c file lines 38 to 48, as the identity augmentation is required in calculating the inverse of any matrix using the Gauss-Jordan method.

The original code also relied on the compiler to convert the integer values to float, as seen in lines 23 to 34 in the unoptimized_code.c file. This could cause unexpected behaviours that we may not be able to see. To rectify this, we replaced the implicit conversion with explicit conversions in our matrix. This change can be seen on lines 12 to 23 in the optimized_code.c file.

Lastly, we neglected to convert our values from float to integers using floating point arithmetic in the unoptimized_code.c file, so in the optimized version, we found the scale factor to be 2^{11} for 12-bit width elements in the range of -1.0 to 1.0. We then embedded the scale factor in every element in the matrix to make the conversion to an integer. Please refer to Figure 1 for an example. The code for the conversion is on lines 32 to 39 in the optimized_code.c file.

4 Compiling and Testing

Compiling and testing this program was fairly streamlined. Because we were remoting into the online UVic-hosted computer system, we didn't have to worry about having any of the programs up-to-date or installed on our machines.

For compiling the program, we used many different flags on the GCC command. This included the static flag, `-static`, to ensure that all libraries were bundled into the executable so that when we ran it, it performed as expected. The STD equals c99 flag, `-std=c99`, which was used to define the standard of C being used in the 99th version. This allowed us to follow the common coding practices that are used today. We also used the `-mfloat-abi` and the `mfpu` flags as described in the lecture documents on how to integrate NEON, and used that system in our optimization. For generating the assembly files, the same flags were used except the static flag was replaced with the dash s flag, `-s`. This allowed us to generate the .s file instead of generating an executable file.

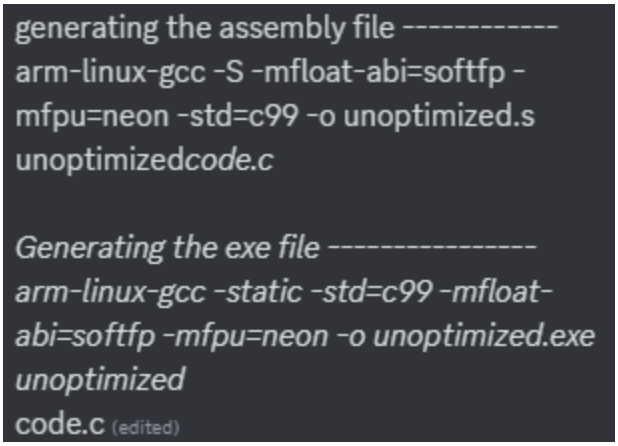


Figure 6. Compilation Commands

For testing the program, we hard-coded in a well and ill-conditioned matrix. We expected that for the unoptimized code, the well and ill-conditioned matrix would perform in approximately the same amount of time. For the optimized version, the well-conditioned matrix would much outperform the ill-conditioned matrix. However, in conducting the actual testing it was discovered that our optimized code did not perform any noticeable amount better than our unoptimized code. Both the well and ill-conditioned matrices had the same run time in the program.

```
1.0000    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300
0.7300    0.9000    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300
0.7300    0.7300    0.8000    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300
0.7300    0.7300    0.7300    0.7000    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300
0.7300    0.7300    0.7300    0.7300    0.6000    0.7300    0.7300    0.7300    0.7300    0.7300
0.7300    0.7300    0.7300    0.7300    0.7300    0.5000    0.7300    0.7300    0.7300    0.7300
0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.4000    0.7300    0.7300    0.7300
0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.3000    0.7300    0.7300
0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.2000    0.7300
0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.7300    0.1000
```

Figure 7. Ill-Conditioned Matrix

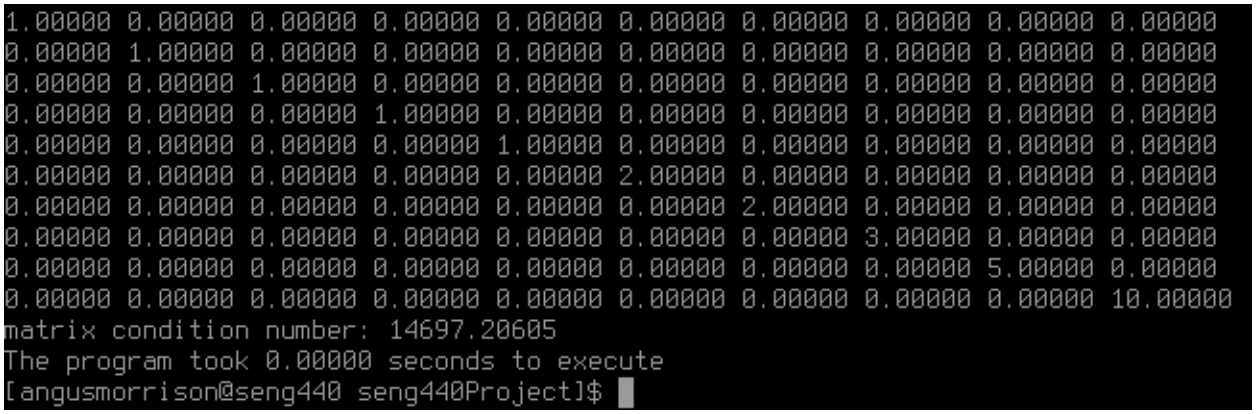


Figure 8. Results of Ill-Conditioned Matrix Inverse (Optimized)

Due to this testing and the compiling that was conducted, we have reasonably determined that our program while handling the optimization methods implemented, and described above has been adequately optimized, due to the optimization performing at the same level for both the well and ill-conditioned matrices, we believe our system to be robust although not fast.

To test our matrix inverse results, we put our matrix into MATLAB and calculated the inverse, and we compared the code output with the output from MATLAB.

5 Performance and Cost Evaluation

For this project, we aimed to achieve some speed up in the optimized code. We specifically expected the vectorization and the register keyword to provide the most optimization, however, we did not achieve any significant performance improvements.

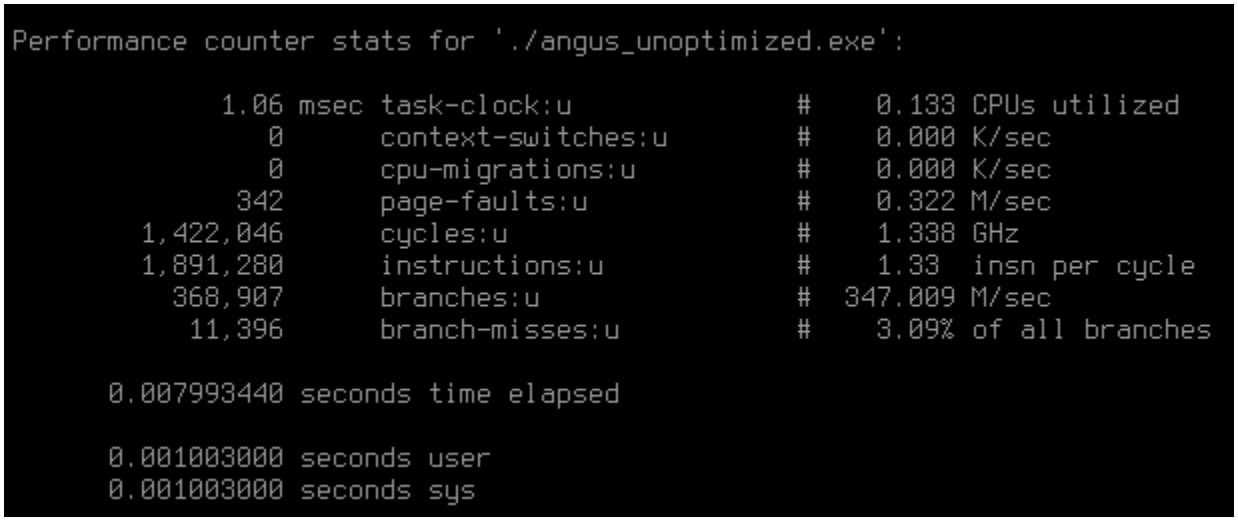


Figure 9. Performance Statistics for Unoptimized Code with Ill-Conditioned Matrix

```
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
```

Figure 10. Performance Statistics for Optimized Code with Ill-Conditioned Matrix

Figure 6 and 7 shows the performance statistics for both optimized and unoptimized code, and we can see that very few improvements were made in the optimized code. We can see that the optimized code had 14,066 fewer cycles than the unoptimized code but this improvement is insignificant in the grand scheme. Similarly, our optimizations did not improve the speed of our program either. As seen in Figures 6 and 7, the unoptimized code executed 1.33 instructions per cycle, while the optimized code executed 1.34 instructions per cycle.

We believe that the code vectorization had the most significant impact on reducing the number of cycles in the optimized code, however, we found that the lack of improvement is due to two factors: the register keyword not working, which also affected the loop decrementation.

```
.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]
```

Figure 8. Unoptimized Assembly: Register Keyword

```
      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]
```

Figure 9. Optimized Assembly: Register Keyword

We observed in the assembly file for the optimized code that the compiler did not take the advice of storing loop counters in registers. This can be seen in Figures 8 and 9. In both Figures, we can see that the loop counter is loaded into the register from memory, then the add/subtract operation is done, and then it is stored back into memory. This also causes the decrement loop to not provide any optimization because the loop counter will have to be loaded from memory into a register again, leading to the inability to check the zero flag, as seen in Figure 9.

6 Discussion

To optimize our program for matrix inversion using Gauss-Jordan for a 10x10 matrix with 12-bit width elements, we implemented four optimization techniques in our code including using the register keyword; decrement loops; code vectorization; and using `memcpy()`. We had expectations that these optimizations would improve the performance of our program, but we found that it made little to no improvements. As stated above, the main challenge we faced was achieving some speed up in our program, but due to the compiler not taking the suggestion to store the loop counters in registers, this led to constant memory accesses when running the code, and our goal with using the register keyword is to minimize these memory accesses when looping through the matrix. This also resulted in the decrement loop not performing the way we were hoping as the loop counter would have to be loaded and stored which led to not being able to check the zero flag. Though those were surprising findings for us, the vectorization using NEON intrinsics did provide the most impact on the minor improvements that we achieved.

Other design changes may have also had a minor impact on the improvements that we were able to achieve. This includes reorganizing our data and appending a hard-coded identity matrix to the test matrix so that we do not need to rely on a for loop to create the identity matrix. We also included other changes to our optimized code that were missed to include in the unoptimized version. This includes explicitly writing floats instead of counting on the compiler to convert integers to float to avoid any unexpected behaviours, and we also found the scale factor and embedded the scale factor into the floating point matrix elements to convert to integers.

For our program, we found the condition number threshold to be less than a condition number of 9; however, this is limited to the optimization that we have implemented in our program. Better optimizations could be possible to be achieved.

7 Conclusion

As stated above, our optimizations for matrix inversion using Gauss-Jordan Elimination on a 10x10 matrix with 12-bit width elements provided little to no improvements to the

performance. For the minor improvements that we achieved, the best optimization technique for us was vectorizing the code using NEON intrinsics which gave us access to vector operation registers. However, this still required us to loop through the matrix 5 times due to our data layout in a 10x20 matrix, where the later ten columns were for the augmented identity matrix. We currently do not suggest any manual optimization, as our results do not show much improvements, but if optimization is required, perhaps further work into optimizing a 10x10 using Gauss-Jordan could look into organizing the data differently and perhaps testing to see if the register keyword would be successful in storing variables in the registers. Furthermore, more optimization using NEON intrinsics to vectorize the code could be valuable for different data organizations of the matrices, and perhaps matrices of different specifications.

References

[1] A. Ltd., “What is RISC?,” Arm, https://www.arm.com/glossary/risc (accessed Aug. 8, 2023).

[2] “Reduced instruction set computer,” Wikipedia, https://en.wikipedia.org/wiki/Reduced_instruction_set_computer (accessed Aug. 8, 2023).

[3] “ARM architecture family,” Wikipedia, https://en.wikipedia.org/wiki/ARM_architecture_family (accessed Aug. 8, 2023).

[4] “M.7 Gauss-Jordan Elimination: Stat Online,” PennState: Statistics Online Courses, https://online.stat.psu.edu/statprogram/reviews/matrix-algebra/gauss-jordan-elimination#:~:text=The%20purpose%20of%20Gauss%2DJordan,the%20bottom%20of%20the%20matrix (accessed Aug. 8, 2023).

[5] “Gaussian elimination,” Wikipedia, https://en.wikipedia.org/wiki/Gaussian_elimination (accessed Aug. 8, 2023).

[6] “Invertible matrix,” Wikipedia, https://en.wikipedia.org/wiki/Invertible_matrix#Applications (accessed Aug. 8, 2023).

[7] “Linear Algebra,” Wikipedia, https://en.wikipedia.org/wiki/Linear_algebra (accessed Aug. 8, 2023).

Appending A: unoptimized_code.c

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

float input_norm;
float output_norm;

void PrintFullMatrix(float matrix[10][20]) {

for (int i = 0; i < 10; i++) {
for (int j = 0; j < 20; j++) {
printf("%f ", matrix[i][j]);
}
printf("\n");
}
}

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

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;
}
}
}

for(int i = 0; i<10; i++){
for(int j = 0; j<10; j++){
input_norm = input_norm + matrix[i][j];
}
}

// Print the whole 10x20 matrix
PrintFullMatrix(matrix);

printf("Finished augmenting the identity matrix\n");

printf("Apply Gauss-Jordan elimination\n");

// 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];
}
}
}

// Print matrix after interchange operations.
printf("\nRows have been interchanged\n");
PrintFullMatrix(matrix);

// This gives reduced row-echelon form
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;
}
}
}
}

// 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;
}
}

printf("\n\nINVERTED MATRIX\n\n");
for (int i = 0; i < 10; i++) {
for (int j = 10; j < 20; j++) {
output_norm = output_norm + matrix[i][j];
printf("%f ", matrix[i][j]);
}
printf("\n");
}

printf("matrix condition number: %.5f\n", input_norm*output_norm);
double time_taken = ((double)(clock()-start_time))/CLOCKS_PER_SEC;
printf("The program took %.5f seconds to execute\n", time_taken);

return 0;
}
```

Appendix B: optimized_code.c

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "arm_neon.h"

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

float input_matrix[10][20] = {
{1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0},
{0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0},
{0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
0.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
0.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
0.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
1.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 1.0, 0.0},
{0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 1.0}
};

register int i = 0;
register int j = 0;
register int k = 0;
register float input_norm = 0;
register float output_norm = 0;
```

```
// float to int conversion
int int_matrix[10][20];
for(i=9; i>=0; i--){
for(j=19; j>=0; j--){
input_norm = input_norm + input_matrix[i][j];
int_matrix[i][j] = (int)(input_matrix[i][j] * 2048); // 2048 is the scale factor
}
}


// 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

// possibly lots of cache misses
float 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));
}
}

// This gives reduced row-echelon form
// fliped this to decrement loops
// made temp a local variable that is not reinitialized each loop iteration
register int div_temp;

uint32x4_t i_vector_row;
uint32x4_t j_vector_row;
register int temp;

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);
}
}
}
}




// Devide each row by the diagnal to get the inverse
for (int i = 0; i < 10; i++) {
int diagonal = int_matrix[i][i];
for (int j = 0; j < 20; j++) {
int_matrix[i][j] = (int_matrix[i][j] * 2048)/ diagonal;
}
}




// convert the int matrix back to a float matrix
// int to float conversion
for(i=9; i>=0; i--){
for(j=19; j>=0; j--){
input_matrix[i][j] = (float)(int_matrix[i][j]/2048); // 2048 is the scale factor
output_norm = output_norm + input_matrix[i][j];
}
}
```

```
// This is just the print of the final matrix
printf("\n\nINVERTED MATRIX\n\n");
for (int i = 0; i < 10; i++) {
for (int j = 10; j < 20; j++) {
printf("%.5f ", input_matrix[i][j]);
}
printf("\n");
}

printf("matrix condition number: %.5f\n", input_norm*output_norm);
double time_taken = ((double)(clock()-start_time))/CLOCKS_PER_SEC;
printf("The program took %.5f seconds to execute\n", time_taken);

return 0;
}
```