Final Report

Seng 440 Matrix Inversion

University of Victoria
Department of Electrical Engineering

August 13, 2021
Submission Date: _______________

Team 15 Members

Amaan Makhani	V00883520	amaanmakhani@gmail.com
Evan Roubekas	V00891470	evan@roubekas.com

Table of Contents

Introduction

This section includes a short description of the domain, performance requirements, enumeration of the contributions, and project organization description.

Project Description

The topic we chose to investigate was matrix inversion. Matrix inversion is a huge topic in the world of embedded systems. It is highly used in real world applications. Here are a few [1]:

- Smaller system of linear equations. In practice large systems of equations use other techniques that are often more efficient. One such technique mentioned was LU decomposition.
- Real time simulations such as computer graphics, particularly in 3D graphics rendering and 3D simulations. These image transformations and manipulations consist of large matrices and need to be fast to produce an effective simulation.
- Data in wireless communications using MIMO (Multiple-Input, Multiple-Output) technology. The MIMO system consists of N transmit and M receive antennas. Unique signals, occupying the same frequency band, are sent via N transmit antennas and are received via M receive antennas. The signal arriving at each receive antenna will be a linear combination of the N transmitted signals forming an $N \times M$ transmission matrix H. It is crucial for the matrix H to be invertible for the receiver to be able to figure out the transmitted information.

These were just a few applications we found of matrix inversion, however, we are aware that many highly used algorithms and techniques also need inverse matrices. Developing a fast and effective algorithm to do so is of utmost importance.

Matrix inversion also faces a few problems. These problems include:

- The algorithms of choice. Cofactor expansion which is discussed in the theoretical background section has a high operation cost and is difficult to execute in parallel. Algorithms like these should be avoided.

- The matrix condition is a description of the magnification factor of an error. A matrix with a small condition number will see a small relative change produce a small relative error in the inverse. The condition number of a matrix must be accounted for to produce an effective solution.
- Computer memory is also another large area for concern as the larger a matrix is the higher chance that cache misses occur is magnified. The more cache misses that occur will result in a less effective solution.

During our topic meeting we were told our specifications were a 10X10 matrix with 13-bit signed integer elements. We plan to compile this program on the virtual machine due to its portability and since the lab computers will likely be very busy as the semester wraps up.

Performance Requirements

We were given a specification of a 10X10 matrix containing 13-bit signed integer elements. This solution was to be optimized using techniques discussed in class. The solution must also contain a test bench with a well-conditioned matrix and an ill-conditioned matrix.

The technique to be used was Gauss-Jordan algorithm with pivoting using integer arithmetic. This solution is thoroughly described in the theoretical background section.

Furthermore a pure-software solution should be compared to a theoretical hardware support for vanishing the column elements.

Contributions

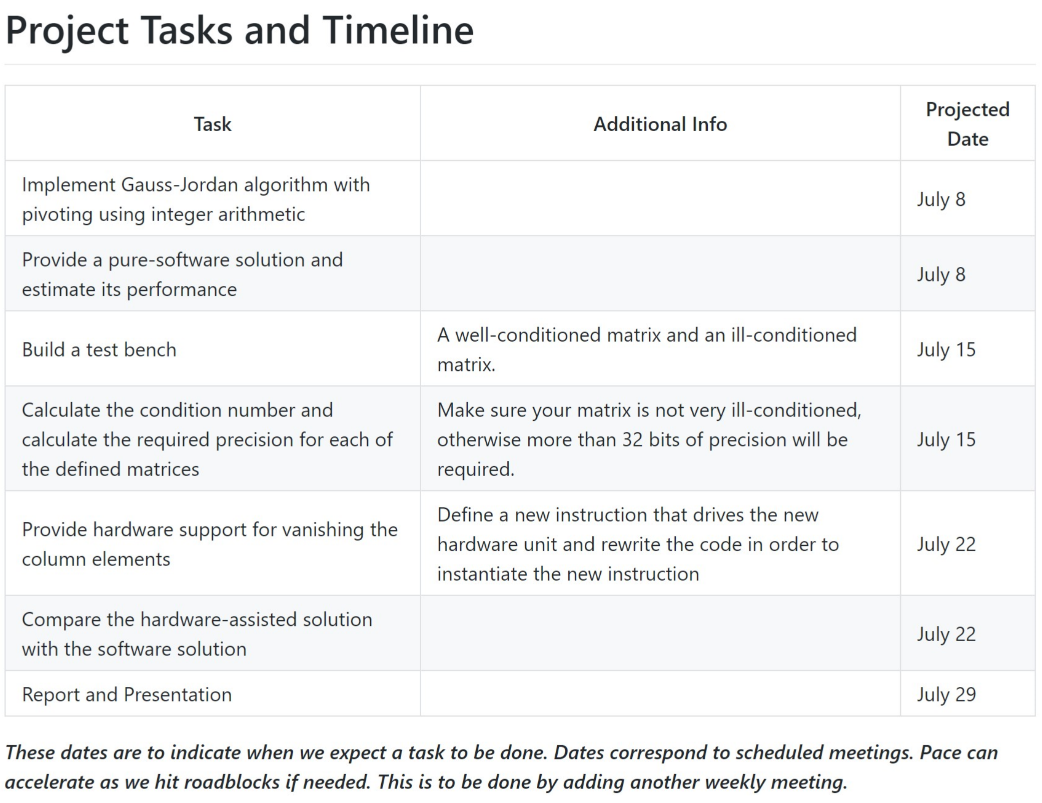
The list of contributions is described in the table below.

Evan	Amaan
Investigation of the Gauss Jordan Algorithm. This included creating miscellaneous functions like calculating the norm and the condition number.	Implementation of the Gauss Jordan Algorithm. This included creating miscellaneous functions like calculating the norm and the condition number.
Investigation of the scale factor and	Investigation of the scale factor and

integer arithmetic. This included meetings and investigation of possible solutions.	integer arithmetic. This included meetings and investigation of possible solutions.
Building the test bench containing both the well and ill conditioned matrices.	Implementation of the software optimization techniques.
Implementation of the Hardware instruction.	Writing of the Project Proposal.
Writing of the Final Report.	Writing of the Final Report.

Project Organization

For project organization, google docs was used to store our written work, while GitHub was used to store our programs. We made use of branches too for certain tasks and the README for schedules and important information. The project was broken down into the requirements and deadlines were set for each requirement. A detailed schedule of deadlines as contained in our GitHub is shown below (figure 1).



Project Tasks and Timeline

Task	Additional Info	Projected Date
Implement Gauss-Jordan algorithm with pivoting using integer arithmetic		July 8
Provide a pure-software solution and estimate its performance		July 8
Build a test bench	A well-conditioned matrix and an ill-conditioned matrix.	July 15
Calculate the condition number and calculate the required precision for each of the defined matrices	Make sure your matrix is not very ill-conditioned, otherwise more than 32 bits of precision will be required.	July 15
Provide hardware support for vanishing the column elements	Define a new instruction that drives the new hardware unit and rewrite the code in order to instantiate the new instruction	July 22
Compare the hardware-assisted solution with the software solution		July 22
Report and Presentation		July 29

These dates are to indicate when we expect a task to be done. Dates correspond to scheduled meetings. Pace can accelerate as we hit roadblocks if needed. This is to be done by adding another weekly meeting.

Figure 1. The project schedule

These dates allowed us to ensure we were on task. However, we ran into problems using integer arithmetic as we struggled to find the implementation location of the respective scale factor. Since this problem occurred we experienced delays in software implementation and worked well into August on solving this issue. We were struggling to line up the respective decimal points since some decimal points had already been scaled while some had not.

These struggles were solved by consulting our Professor and other colleagues.

Theoretical background

The theoretical background described in this section was gathered from the slide deck on our project [2].

The condition number of a matrix measures how the error in the input data affects the computed answer. For matrix inversion the condition number κ is $||A||\cdot||A^{-1}||$, where $||A||$ and $||A^{-1}||$ is the matrix norm for the original matrix and the inverse matrix respectively.

The norm is calculated as follows:

$$||A|| = \max_i \sum_j |a_{ij}|$$

Figure 2. Norm calculation

This means that the norm of a matrix is the maximum value of the row. Where all the absolute values of the row elements are used.

A well conditioned matrix is where κ is small relative to one . When the condition number is small a small relative change produces a small relative error in the inverse. When the condition number is large a small relative change produces a large relative error in the inverse.

A few methods can be used to find the inverse of a matrix:

Cofactor expansion is one method. Using this method we calculate a matrix of cofactors as shown below. |A| represents the determinant of the matrix, Cji is the cofactor matrix, and T is the transpose of a matrix. For any method if |A| = 0 then the inverse cannot exist.

$$A^{-1} = \frac{1}{|A|}(C_{ij})^T = \frac{1}{|A|}\begin{pmatrix} C_{11} & C_{21} & \cdots & C_{j1} \\ C_{12} & \ddots & \vdots & C_{21} \\ \vdots & \cdots & \ddots & \vdots \\ C_{1i} & \cdots & \cdots & C_{ji} \end{pmatrix}$$

Figure 3. Cofactor expansion

The cofactor Cij of A is defined as $(-1)^{i+j}$ multiplied by the minor M_{ij} of A. Where minor M_{ij} of A is the determinant of the smaller matrix that results from A by removing the i-th row and j-th column.

For example to calculate the determinant:
If A is a 1-by-1 matrix, then $|A| = A_{1,1}$
If A is a 2-by-2 matrix, then $|A| = A_{1,1}A_{2,2} - A_{2,1}A_{1,2}$
If A is a 3-by-3 matrix, then $|A| = A_{1,1}A_{2,2}A_{3,3} + A_{1,3}A_{3,2}A_{2,1} + A_{1,2}A_{2,3}A_{3,1} - A_{3,1}A_{2,2}A_{1,3} - A_{1,1}A_{2,3}A_{3,2} - A_{1,2}A_{2,1}A_{3,3}$

As you can see above, calculating the determinant is a computationally-intensive task as many calculations need to be completed, and many operands are used. These operands can get highly expensive in terms of computer processing cycles. If you were to add hardware/firmware support this computation intensity is not reduced. Due to these reasons cofactor expansion is too expensive and not the best option.

Another technique is matrix inversion by Gauss-Jordan elimination. In this method a matrix is converted into a system of equations. We then vanish all matrix elements but one keep one element per column.

This process involves manipulation of the identity matrix at the same time as manipulating the matrix to be solved. The identity matrix is one where the one's are located on the diagonal. The problem here is we increase cache misses since both matrices are manipulated at the same time. This also makes this technique slower due to this issue. This method also fails when an attempted division by zero occurs.

To solve this issue Gauss-Jordan elimination with pivoting is used. To avoid the instability of the standard Gauss-Jordan elimination, the largest element along the column is declared the pivot. A side effect of this is extra calls to memory due to finding the largest column element. Instead of looking through rows which are cached the CPU must access a new row every time. Pivoting does have it's cons but it has more benefits when compared to other methods available to us.

To represent our matrices with 13-bit persion a scale factor has to be added. This scale factor aims to first remove the fractional composition of an element and ensure precision isn't lost in calculations. Small differences will occur due to scaling but will be small when a well conditioned matrix is used, this is called numerical instability.

Since Gaussian elimination with pivoting is the best solution it was implemented for our solution. This technique was implemented as follows.

Any operations done on the matrix are also done to the Identity matrix. The identity matrix after all the calculations is called an augmented matrix. This is the inverse matrix. The goal is to turn the original matrix into reduced row echelon form. The reduced row echelon form is when :

- It is in row echelon form. Row echelon form is when the first non-zero number from the left is always to the right of the first non-zero number in the row above. To be in row echelon form also means that rows consisting of all zeros are at the bottom of the matrix.
- The leading entry in each nonzero row is a 1 (called a leading 1).
- Each column containing a leading 1 has zeros in all its other entries.

Gauss-Jordan elimination with pivoting works as follows:

1. Find the largest element in the column and swap that row to the diagonal

2. Get a "1" on the diagonal by dividing the row by the element on the diagonal
3. Then we get "0" in the rest of the column, this is done by
   The row element - (column element to eliminate * corresponding row element from the row the one was produced in)

This is done for each column. When implementing our algorithm we followed an algorithmic process found online [3]. The process above is shown below:

$$\begin{array}{l} \left[\begin{array}{ccc|ccc} 9 & 13 & 3 & 1 & 0 & 0 \\ 5 & 14 & 6 & 0 & 1 & 0 \\ 7 & 8 & 11 & 0 & 0 & 1 \end{array}\right] \begin{array}{l} \text{Row[1]} \\ \text{Row[2]} \\ \text{Row[3]} \end{array} \end{array}$$

Figure 4. Matrix after step 1

Divide Row [1] by 9 (to give us a "1" in the desired position)

$$\left[\begin{array}{ccc|ccc} 1 & 1.4444 & 0.3333 & 0.1111 & 0 & 0 \\ 5 & 14 & 6 & 0 & 1 & 0 \\ 7 & 8 & 11 & 0 & 0 & 1 \end{array}\right] \begin{array}{l} \text{Row[1]} \\ \text{Row[2]} \\ \text{Row[3]} \end{array}$$

Figure 5. Matrix after step 2

Row[2] − 5 × Row[1] (to give us 0 in the desired position):

$5 - 5 \times 1 = 0$
$14 - 5 \times 1.4444 = 6.7778$
$6 - 5 \times 0.3333 = 4.3333$
$0 - 5 \times 0.1111 = -0.5556$
$1 - 5 \times 0 = 1$
$0 - 5 \times 0 = 0$

Figure 6. Matrix operations executed in step 2

$$\left[\begin{array}{ccc|ccc} 1 & 1.4444 & 0.3333 & 0.1111 & 0 & 0 \\ 0 & 6.7778 & 4.3333 & -0.5556 & 1 & 0 \\ 7 & 8 & 11 & 0 & 0 & 1 \end{array}\right] \begin{array}{l} \text{Row[1]} \\ \text{Row[2]} \\ \text{Row[3]} \end{array}$$

Figure 7. Matrix after step 3

Design Process

To design our solution we broke our process into numerous steps:

- To attack this problem we are first implementing Gauss Jordans algorithm for matrix inversion without pivoting. This allowed us to start the process of matrix inversion. In this step we also implemented a norm function to allow us to calculate the condition number of the matrix. Due to fractions in the inverse we started using a float data type to ensure the algorithm's correctness. We started with a 4x4 matrix for easier debugging.
- We then created a well and ill conditioned matrix that we had a known solution for. This allowed us to continue testing our solution for breaking changes.
- We then added in row pivoting as all this required was a swap to have the largest column element and it's row the next non reduced row. This allowed us to eliminate the largest column values first and avoid division by zeros.
- We then had our software solution and targeted it's bottlenecks. To do this we generated a baseline to refer our optimizations to. Using the optimizations in class we created a series of tests to choose the best optimization technique/techniques.
- Since we ran into issues implementing our scale factor we added that in just before generating our hardware assisted code. This did mean re-running previous tests but this allowed us to continue working. This scale factor allowed us to use integer arithmetic.
- We finished by adding in our hardware instruction and comparing the effectiveness of this solution versus the software solution.

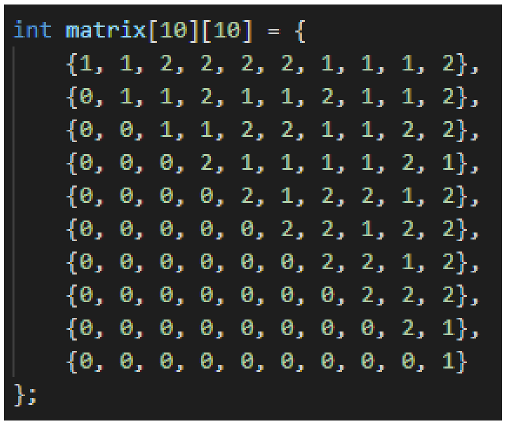
Furthermore, the interactive process is shown below through archived code.

1. Gaussian_Elimination_with_Pivoting, located in the iterations folder
2. Gaussian_Elimination_with_Scaling_and_Pivoting, located in the iterations folder
3. Gaussian_Elimination_without_Pivoting, located in the iterations folder
4. Unoptimized__with_Integer_Arithmetic, located in the iterations folder
5. not_optimized_without_hardware
6. optimized
7. optimized_with_hardware

Test Bench

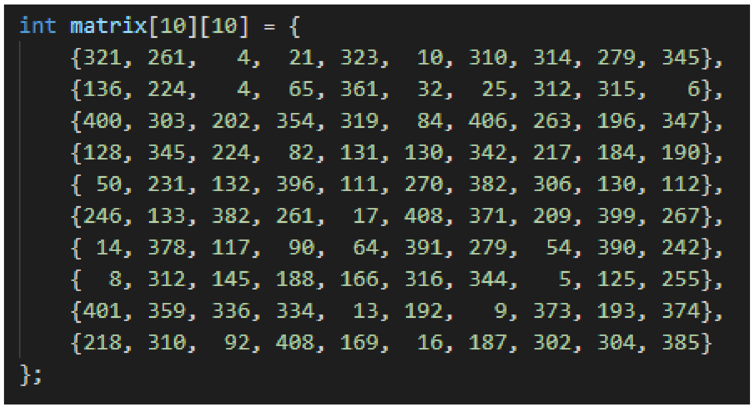
After finishing the implementation of our inversion algorithm using the background described previously we needed to find a well conditioned and ill conditioned matrix that we could use for our test bench. Finding these matrices involves a lot of trial and error, as there is not an obvious best method to find matrices that are conditioned how you want. That being said an insight that helped to find a well conditioned matrix was that the identity matrix had a norm of 1, therefore matrices similar to the identity matrix will be better conditioned.

Ultimately we ended up choosing these two matrices, see figure 8 for the well conditioned matrix and figure 9 for the ill conditioned matrix. These matrices have condition numbers of 15 and 5748 respectively. Therefore with 13 bits of precision we would need an extra 4 bits for the well conditioned matrix and an extra 12 bits for the ill conditioned matrix. We get these numbers by raising 2 to a power until that number is greater than the condition number. For example 2^4 = 16 > 15 but 2^3 = 8 < 15 so therefore 4 bits are needed for the well conditioned matrix.



```
int matrix[10][10] = {
    {1, 1, 2, 2, 2, 2, 1, 1, 1, 2},
    {0, 1, 1, 2, 1, 1, 2, 1, 1, 2},
    {0, 0, 1, 1, 2, 2, 1, 1, 2, 2},
    {0, 0, 0, 2, 1, 1, 1, 1, 2, 1},
    {0, 0, 0, 0, 2, 1, 2, 2, 1, 2},
    {0, 0, 0, 0, 0, 2, 2, 1, 2, 2},
    {0, 0, 0, 0, 0, 0, 2, 2, 1, 2},
    {0, 0, 0, 0, 0, 0, 0, 2, 2, 2},
    {0, 0, 0, 0, 0, 0, 0, 0, 2, 1},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 1}
};
```

Figure 8. Well Conditioned Matrix with a condition number of 15



```
int matrix[10][10] = {
    {321, 261,   4,  21, 323,  10, 310, 314, 279, 345},
    {136, 224,   4,  65, 361,  32,  25, 312, 315,   6},
    {400, 303, 202, 354, 319,  84, 406, 263, 196, 347},
    {128, 345, 224,  82, 131, 130, 342, 217, 184, 190},
    { 50, 231, 132, 396, 111, 270, 382, 306, 130, 112},
    {246, 133, 382, 261,  17, 408, 371, 209, 399, 267},
    { 14, 378, 117,  90,  64, 391, 279,  54, 390, 242},
    {  8, 312, 145, 188, 166, 316, 344,   5, 125, 255},
    {401, 359, 336, 334,  13, 192,   9, 373, 193, 374},
    {218, 310,  92, 408, 169,  16, 187, 302, 304, 385}
};
```

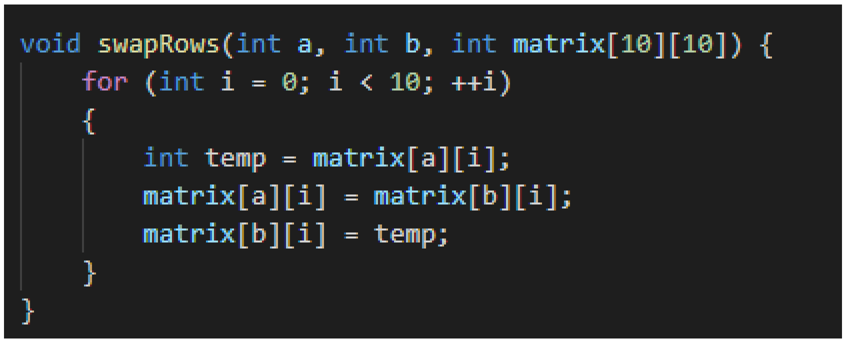
Figure 9. Ill Conditioned Matrix with a condition number of 5748

Optimization Techniques

In this section we will discuss the various software optimization techniques available to us, why we chose to implement some, and why we found others to not be as applicable.

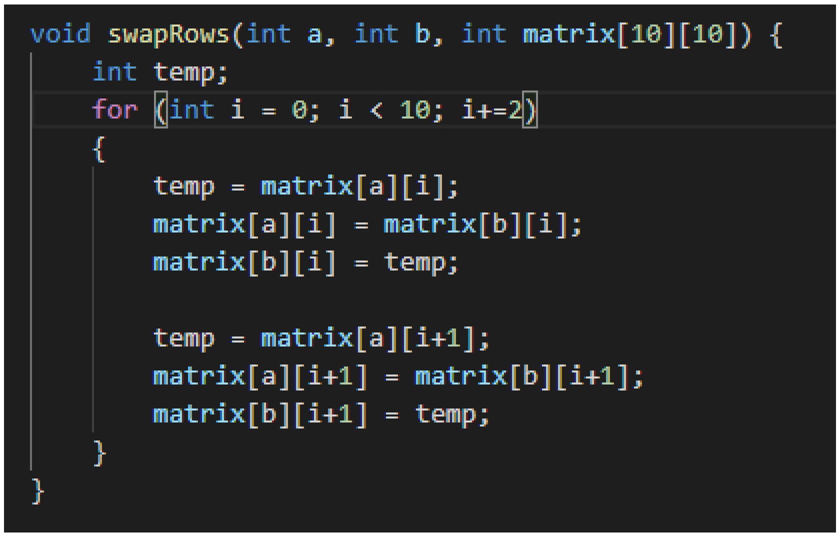
Software Pipelining / Loop Unrolling

For optimizing the multiple for loops in our code we chose to use loop unrolling over software pipelining to keep the simplicity of the program. A big downside to loop unrolling is the increased memory usage, however in our case as seen below in figure 10 and 11 the majority of our loop unrolling throughout our code involves the matrix and inverse matrices which already have memory allocated. Therefore by using loop unrolling as a technique we dodge the increased memory usage while still decreasing the overall code size.



```
void swapRows(int a, int b, int matrix[10][10]) {
    for (int i = 0; i < 10; ++i)
    {
        int temp = matrix[a][i];
        matrix[a][i] = matrix[b][i];
        matrix[b][i] = temp;
    }
}
```

Figure 10. The swapRows function without loop unrolling



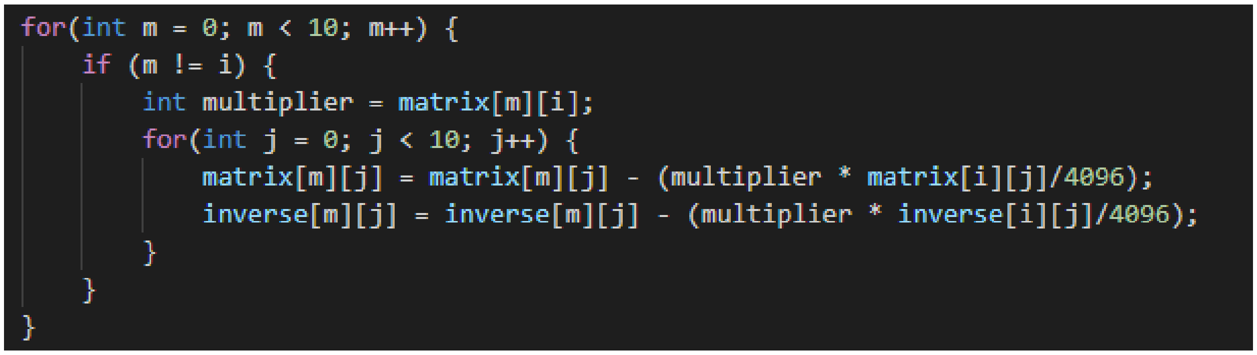
```
void swapRows(int a, int b, int matrix[10][10]) {
    int temp;
    for (int i = 0; i < 10; i+=2)
    {
        temp = matrix[a][i];
        matrix[a][i] = matrix[b][i];
        matrix[b][i] = temp;

        temp = matrix[a][i+1];
        matrix[a][i+1] = matrix[b][i+1];
        matrix[b][i+1] = temp;
    }
}
```

Figure 11. The same function but with loop unrolling implemented

Operator Strength Reduction

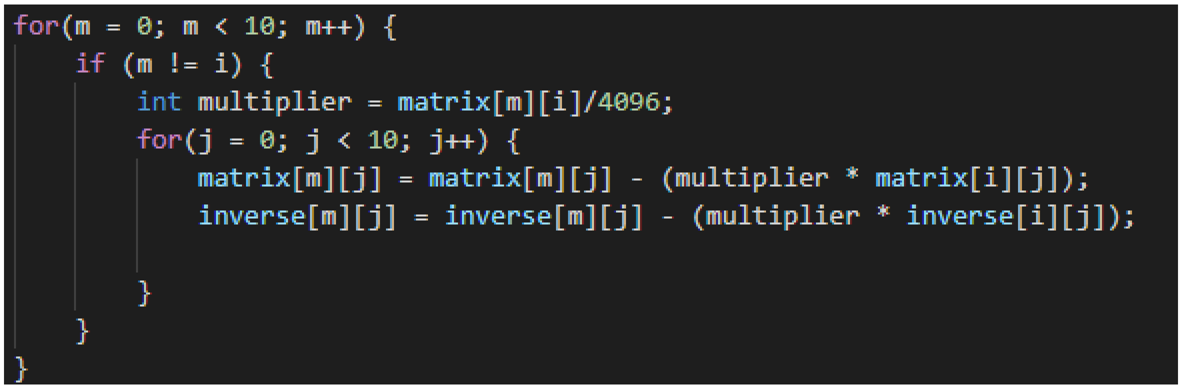
Operator strength reduction involves moving or replacing a costly operator with a cheaper one. The goal is to eliminate divisions as they are costly operations (around 10 cycles). In our code we have a double for loop block that is used for vanishing column elements (see figure 12), where division occurs 200 times (10*10 for both the matrix and inverse matrices).



```
for(int m = 0; m < 10; m++) {
    if (m != i) {
        int multiplier = matrix[m][i];
        for(int j = 0; j < 10; j++) {
            matrix[m][j] = matrix[m][j] - (multiplier * matrix[i][j]/4096);
            inverse[m][j] = inverse[m][j] - (multiplier * inverse[i][j]/4096);
        }
    }
}
```

Figure 12. The for loop for column vanishing without operator strength reduction

By implementing operator strength reduction we are able to reduce the number of divisions from 200 to 10 by applying the division directly to the multiplier variable inside the first for loop (see figure 13). Therefore inside the second for loop only a multiplication needs to occur, this greatly reduces the number of cycles needed for the execution of this -algorithm.



```
for(m = 0; m < 10; m++) {
    if (m != i) {
        int multiplier = matrix[m][i]/4096;
        for(j = 0; j < 10; j++) {
            matrix[m][j] = matrix[m][j] - (multiplier * matrix[i][j]);
            inverse[m][j] = inverse[m][j] - (multiplier * inverse[i][j]);

        }
    }
}
```

Figure 13. The same for loop but with operator strength reduction

However this optimization method unfortunately only worked with the ill conditioned matrix, as implementing it with the well conditioned method produced the wrong results. Later in the report when comparing assembly cycles there will be a discrepancy between the optimized well.s and ill.s files for this express reason.

Removing function calls

While during the implementation of this algorithm we tried to avoid using functions altogether to remove the overhead of calling them when executing. However since we need to find the norm of the matrix before and after inverting the matrix we decided to make a function for that to remove code reuse. Furthermore the norm function is too complicated to turn into a macro so it is the exception here when removing function calls.

Efficient Loops

When optimizing loops outside of loop unrolling we tried to optimize the loop efficiency by optimizing the loop counters and taking a look at the exit conditions. For loop counters we initialize them as register unsigned ints (the register keyword will be discussed further in the optimization section). The ints are unsigned because they should never be negative (we have no decrementing loops) and they are not short ints specifically because the ARM performs only 32-bit addition. So by making the ints short and dedicating only 16-bits is counterproductive as the compiler emulates the 16-bit short int into a 32-bit int for the operations. Therefore as discussed in lectures it is a good programming technique to use int and not short int types for loop counters. For optimizing the loop efficiency we were not able to do much past changing the loop counter. As our exit conditions were not multiples of 2 we could not do bitwise ors or simplify the exit in a succinct way. The same is true for incrementing the loop.

Restrict and no_alias

Since we do not have any functions outside of the norm, and the norm only takes one argument there is no need to implement the restrict keyword. The same is true for the no_alias pragma.

Function Inlining

However even though we cannot restrict or no_alias our function parameters we are able to optimize the norm function by adding the keywords static inline to it. By inlining the function we optimize the program by reducing the number of possible cache misses and improves the memory location of the function within the code.

Register

For temporary smaller variables like the loop counters and integers that hold the norm data we use the register keyword to encourage the compiler to keep these values stored in registers. This aims to reduce the memory overhead required by the program. However, the register keyword is just a suggestion so we cannot enforce the compiler to store this data in the registers

Short Unsigned Ints

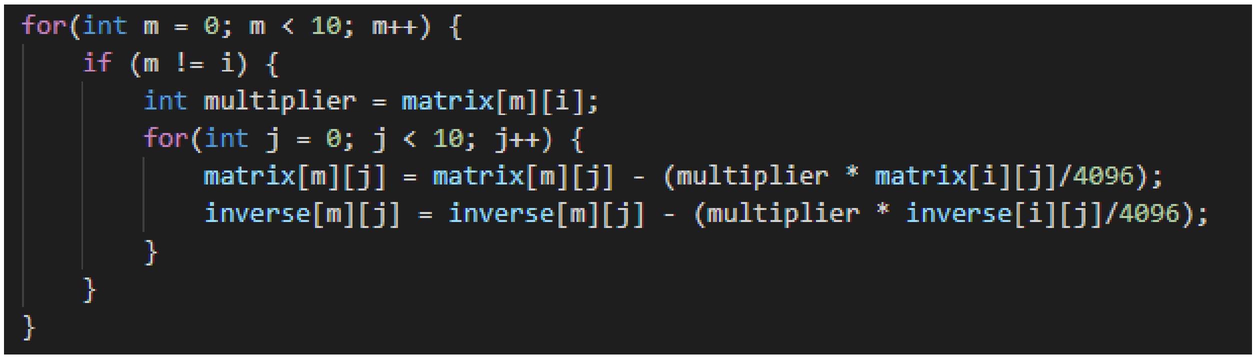
Like with the loop counters we use short unsigned int to define data that we know will be smaller and not negative, an example being the variable holding the norm data. We do this in an attempt to again optimize memory usage and cycle overhead when executing the algorithm.

Dead Code Elimination

Lastly in an attempt to optimize our program we identified and eliminated any dead code. In fact we applied this to our "not optimized" code as well as we believe it to be a good coding practice optimization aside. This includes any code that is not reached because of a conditional or lines after any returning. By eliminating this code not only do we simplify the program but again if not eliminated by the compiler this also reduces the cycles in the compiled assembly.

Hardware Inlining

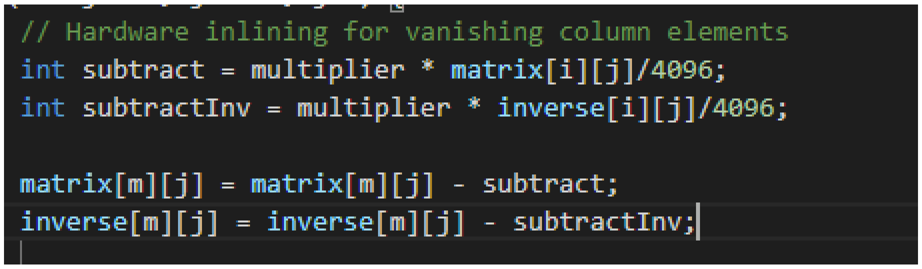
As specified in the project requirements we were to provide hardware support for vanishing column elements using an inline instruction and then re-write our code to implement this new instruction. Originally our algorithm vanishes column elements in a double for loop seen below in figure 14.



```
for(int m = 0; m < 10; m++) {
    if (m != i) {
        int multiplier = matrix[m][i];
        for(int j = 0; j < 10; j++) {
            matrix[m][j] = matrix[m][j] - (multiplier * matrix[i][j]/4096);
            inverse[m][j] = inverse[m][j] - (multiplier * inverse[i][j]/4096);
        }
    }
}
```

Figure 14. Non optimized column vanishing

Our goal is to use hardware support to implement an instruction that achieves that subtraction that essentially vanishes the column element. In other words we need to implement an assembly instruction using the __asm__ keyword to do the subtraction seen in figure 15.



```
// Hardware inlining for vanishing column elements
int subtract = multiplier * matrix[i][j]/4096;
int subtractInv = multiplier * inverse[i][j]/4096;

matrix[m][j] = matrix[m][j] - subtract;
inverse[m][j] = inverse[m][j] - subtractInv;
```

Figure 15. Same code as above rewritten with variables

Performance/Cost Evaluation

After implementing the hardware inlining and software optimizations we identified four scenarios we wanted to compare, totalling to 8 simulations (each scenario containing the ill and well conditioned cases).

1. Not optimized software solution

The first scenario is matrix inversion with a pure software solution without any optimizations. After compiling the ill.c and well.c files into assembly we get these results:

Ill conditioned matrix (ill.s): 646 assembly instructions
Well conditioned matrix (well.s): 646 assembly instructions

2. Optimized software solution

The second scenario is matrix inversion with the software optimization discussed in the earlier sections. After compiling the ill.c and well.c files into assembly we get these results:

Ill conditioned matrix (ill.s): 585 assembly instructions
Well conditioned matrix (well.s): 620 assembly instructions

As discussed in software optimization, operator strength reduction could not be implemented for the well conditioned matrix hence the higher instruction count we see here.

3. Not optimized solution with hardware inlining

The third scenario is matrix inversion with a pure software solution without any optimizations and hardware support. After compiling the ill.c and well.c files into assembly we get these results:

Ill conditioned matrix (ill.s): 632 assembly instructions
Well conditioned matrix (well.s): 632 assembly instructions

4. **Optimized solution with hardware inlining**

The fourth scenario is matrix inversion with a pure software solution with optimizations and hardware support. After compiling the ill.c and well.c files into assembly we get these results:

Ill conditioned matrix (ill.s): 797 assembly instructions
Well conditioned matrix (well.s): 868 assembly instructions

After some testing we are not quite sure why the optimized solution with the hardware inlining increased the number of instructions so drastically. With more time for the project we would like to investigate this further, perhaps there is a conflict with some of the optimizations implemented and how the asm inlining works.

Based on these four scenarios and the multiple simulations run, the optimized software solution produced the best results in that there are the least amount of assembly instructions required to run the program.

Conclusion

In conclusion, matrix inversion has huge application in the real world and optimizing a solution to obtain an inverse matrix is very important. After the analysis of various techniques to obtain an inverse matrix we were to implement Gausian elimination with pivoting as it was a better solution than most. We were able to find good examples of an ill conditioned and well conditioned matrix to use in our test bench and as a base standard to run our programs against. Our ill conditioned matrix had a condition number of 5748 and needed 12 extra bits to estimate whereas our well conditioned matrix had a condition number of 15 and only needed 4 extra bits. A software solution was optimized with a test bench containing a well and an ill conditioned matrix. Various software optimization techniques discussed in class were used including loop unrolling, operator strength reduction, and function inlining among others. It was observed when compiling the code to assembly that the optimization had the desired effect and reduced the number of assembly instructions for both the ill and well conditioned matrices. Furthermore a requirement of this project was to use hardware inlining to complete the vanishing column element operation of our algorithm. We were able to implement this using an asm assembly inline function, and as shown in the performance evaluation section we showed that even without software optimizations the use of hardware support was able to reduce the number of assembly instructions. Unfortunately as also noted in that section the use of hardware support and software optimization together produced the opposite effect, this was something we were not expecting but was interesting to observe nonetheless.

We discovered many new techniques and made critical observations when exploring this topic. We hope to take away our learning and bring it with us in future endeavors.

Bibliography

[1]. “Invertible matrix,” *Wikipedia*, 04-Aug-2021. [Online]. Available: https://en.wikipedia.org/wiki/Invertible_matrix#Applications. [Accessed: 12-Aug-2021].

[2]. M. Sima, “Embedded_Systems_Slides_WRAPON_lesson_111,” in *SENG 440 Embedded Systems*.

[3]. M. Bourne, “Inverse of a Matrix using Gauss-Jordan Elimination,” *intmathcom RSS*. [Online]. Available: https://www.intmath.com/matrices-determinants/inverse-matrix-gauss-jordan-elimination.php. [Accessed: 12-Aug-2021].