**Matrix Inversion Project Report**

**SENG 440 - Embedded Systems**

**Nathan Tutak V00920892**

**Ewan Morgan V00948587**

**Aug 14th, 2022**

## Table of Contents

[**Table of Contents**](#_3gzx2p74ro86) **2**

[**1 Introduction**](#_3znysh7) **3**

[**1.1 Contributions**](#_uxjxntbt88o7) **3**

[**1.2 Build information**](#_k9q13a9ytj0j) **3**

[**2 Theoretical Background**](#_2s8eyo1) **4**

[**3 Design**](#_jptv0ecpej8e) **4**

[**3.1 General algorithm**](#_34yim17clvo3) **4**

[**3.1.1 Augment Function**](#_m7nroz3h6x9k) **5**

[**3.1.2 Elimination Function**](#_q08qyyagmdf5) **5**

[**3.1.3 Supporting Functions**](#_rpuhjl5ay23u) **6**

[**3.2 Optimizations**](#_prirvoluewzk) **6**

[**3.2.1 CPU Registers**](#_mpwh42yt65rn) **6**

[**3.2.2 Loop Fission**](#_migqdiozf57n) **6**

[**3.2.3 Vector operations**](#_e959b9ddj020) **7**

[**3.3 Hardware assisted solution**](#_auyqwse04m4h) **9**

[**4 Discussion**](#_bozi2dvvrgb0) **9**

[**4.1 Results**](#_840xqqnvn39j) **9**

[**4.2 Performance analysis**](#_g94pn7wmo8qz) **10**

[**4.2.1 CPU Registers**](#_z1i6bucnvs3c) **10**

[**4.2.2 Loop Fission**](#_qvys0kio74f9) **11**

[**4.2.3 Vector operations**](#_sn2jlhm9xtke) **11**

[**4.2.4 Hardware assist**](#_tu3wq6zgmr92) **12**

[**4.3 General discussion**](#_wh3rwnhc92du) **12**

[**5 Conclusion**](#_3nmmhjcxmp3c) **12**

[**6 References**](#_y7toku2u0roj) **13**

[**7 Appendices**](#_hdti2y9cyc77) **14**

[**Appendix A - General Inversion C code**](#_r5xvvfesexy) **14**

[**Appendix B - Optimization code sections**](#_se68pwguv9ab) **18**

### 

## Table of Figures

[Figure 1: Elimination Function 7](#_Toc111374457)

[Figure 2: Swap Function code snippet 8](#_Toc111374458)

[Figure 3: Multiply vector by scalar 8](#_Toc111374459)

[Figure 4: Vector addition 9](#_Toc111374460)

[Figure 5: Vector operations performed for a single row 10](#_Toc111374461)

[Figure 6: Hardware assist c code function 10](#_Toc111374462)

[Table 1: Performance test results 11](#_Toc111374463)

# 1 Introduction

This report discusses inverting a matrix using the Gauss-Jordan elimination method with full pivoting using integer arithmetic. The objective is to compare a pure software solution and a hardware-supported version. The pure software solution requires efficient optimization to reduce the run-time of the algorithm and a method to estimate the performance of the code. It was important to create a test bed with various matrices to analyze the functionality of the code.

Our software solution will take an input of 12-bit word length elements in a matrix of up to 10x10 in size. The goal of our software solution is to calculate the inverse of up to a 10x10 matrix that fits within our specified condition number of 2^14. For each matrix, the condition number was calculated to measure how the error in the input data affects the computed value. In order to accommodate the condition number, an extra 14-bits were used to store the output. Lastly, it's important to ensure that the input matrix is within the specified condition number to avoid requiring the precision to be more than 32 bits.

The hardware solution for vanishing the column elements requires defining a new instruction to drive the hardware unit and rewriting the code to instantiate the added instruction. To run the solutions, we accessed the virtual arm machine through a virtual Linux operating system.

## 1.1 Contributions

Nathan Tutak

* General C code development
* Neon vector operation development
* Report writing and editing
* Performance testing and analysis

Ewan Morgan

* Condition number calculations
* Test bed creation
* QA testing
* Report writing and editing

## 1.2 Build information

The following commands were used to compile the executables:

arm-linux-gcc inverse.c -std=c99 -o inverse -O3

/opt/arm/4.3.2/bin/arm-linux-gcc -mfloat-abi=softfp -mfpu=neon -static -O3 vector\_inverse.c

The second command only applies to the software version using neon intrinsics for the purposes of vector operations. The code was tested on the ARM virtual machine provided by the professor, which is a Fedora 29 ARMv7 specification machine.

# 2 Theoretical Background

Various algorithms which calculate the inverse of a matrix encounter numerous issues involving a huge number of operations and potential instability. The cofactor expansion algorithm for matrix inversion uses a recursive method to find the determinant of a matrix. Since the method is recursive, for larger matrices, it becomes increasingly inefficient because it requires a massive amount of operations.

Due to the inefficiencies of the cofactor expansion method, it is typically avoided. However, alternative inversion techniques are based on matrix factorization instead of recursion. The standard Gauss-Jordan elimination algorithm uses matrix factorization to solve systems of linear equations to find the inverse of any invertible matrix. With the goal of sequentially eliminating all the variables in the matrix except one per column. The standard Gauss-Jordan elimination method encounters issues with memory footprint which may increase cache misses. Unfortunately, this algorithm can fail entirely when it attempts to divide by zero. It may expose the parallelism of the compiler, decreasing the run-time duration.

However, the standard Gauss-Jordan algorithm has an increased chance of numerical instability. To avoid the instability, an alternative is the Gauss-Jordan elimination method with row interchanges. This method picks the largest element along a column declaring it the pivot and stores the matrix in a cache row-wise. The main drawback is it may generate a cache miss from accessing the matrix column-wise. In the first step, the Gauss-Jordan algorithm picks a pivot row based on the max element in the column and divides it by the first element in the pivot row. Then, the algorithm adds the first row to the remaining rows such that the coefficients in the first column become zeros. To achieve this, on the i-th row, we must swap the i-th row with the pivoting row.

# 3 Design

The following section discusses the general algorithm and design choices made in our project, as well as describe the optimizations performed and the design of the hardware assisted solution.

## 3.1 General algorithm

To commence the software implementation portion, we designed a matrix inversion algorithm using Gauss-Jordan elimination without row interchanges. This allowed us to get the basic idea of the algorithm without doing the difficult row interchanges. We created various functions to support the algorithm, many of which carried over to the final implementation. The following section will discuss each part of the algorithm and the design choices made.

### 3.1.1 Augment Function

The augment function was created to augment the input matrix with the identity matrix. The values in the identity matrix are scaled by a specified shift amount in order to implement the fixed point arithmetic. The scale factor used for the fixed point arithmetic was chosen to be 2^20, which allowed for adequate precision for the final output. The code for the augment function can be found in appendix A.

### 3.1.2 Elimination Function

The elimination function performs the actual Gauss-Jordan Elimination algorithm. Starting at the first element in the first row, the function calls a supporting function to find the maximum element in the column and returns the index. The function then swapped the current row with the pivot row through another supporting function. This achieves full pivoting, which ensures the algorithm never divides by zero on an invertible matrix. Once the rows are swapped, the function eliminates the elements in the row and correspondingly adjusts the identity matrix. The elimination function is shown in the following code snippet, **Figure 1**.

Figure 1: Elimination Function

|  |
| --- |
| int\_type first;  int\_type second;  for(int k = 0; k < size; k++) {  // finding the pivot row  int pivot\_index = max\_column(k);  if(k < pivot\_index)  swap(k, pivot\_index);  first = augmented[k][k];  for(int i = 0; i < size; i++) {  second = augmented[i][k];  if(second == 0) continue; // skip rows where its already 0  if(i != k) {  for(int j = 0; j < size\*2; j++) {  //do the elimination and subtracts across the row  augmented[i][j] -= second\*augmented[k][j] / first;  }  }  }  } |

### 

### 3.1.3 Supporting Functions

The swap function was designed to perform row interchange on the pivot row throughout the algorithm. The max column function scanned the matrix column-wise and found the largest element in the matrix for a given column and returned the index. This allowed us to determine the pivot for this column. The output matrix can then be displayed using a variety of print functions that were implemented to serve different purposes. The most important part is to normalize the matrix during this step since this shows the actual value of the inverted matrix while also having the effect of removing the scale factor.

The last supporting function was used for measuring the performance of the various optimizations using the timer library's clock\_gettime function, along with an additional sub timespec function which allowed us to calculate the difference between the start of the program and the end. This method allows for a precise measurement of the execution time.

## 3.2 Optimizations

The following section covers the software optimization techniques that were employed to improve the performance of our project.

### 3.2.1 CPU Registers

The first optimization was to simply use the register keyword to tell the compiler to store variables in CPU registers instead of memory, allowing for faster access. By using registers to store frequently used values, we were able to optimize the access time for the variables and help speed up the execution time. In our solution, the max and index variables were accessed frequently in the algorithm for finding the maximum value in a column. By utilizing the CPU registers we were able to improve the performance of this function.

### 3.2.2 Loop Fission

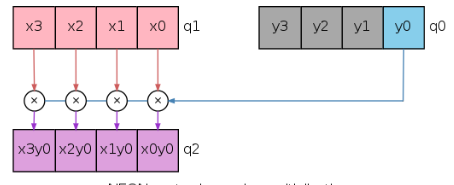
Another optimization that we were able to use is the technique of loop fission. This involves breaking a single loop that performs several distinct tasks into multiple loops over the same index range. The goal of loop fission is to achieve better utilization of locality of reference by accessing the same set of memory locations. We used this optimization technique in the row swapping function, where the loop was broken down into 3 separate loops. In the following code snippet, **Figure 2,** the single loop version is on the left with the loop fission version on the right.

|  |  |
| --- | --- |
| for(int i = 0; i < size\*2; i++) {  temp = a[row1][i];  a[row1][i] = a[row2][i];  a[row2][i] = temp;  } | for(int i = 0; i < size\*2; i++) {  temp[i] = a[row1][i];  }  for(int i = 0; i < size\*2; i++) {  a[row1][i] = a[row2][i];  }  for(int i = 0; i < size\*2; i++) {  a[row2][i] = temp[i];  } |

Figure 2: Swap Function code snippet

### 3.2.3 Vector operations

The most significant software optimization was the use of vector operations. Neon intrinsics are an implementation of ARM Single Instruction Multiple Data. We can use the vector operations they define to complete arithmetic operations on multiple elements of the matrix at the same time. This presents a significant improvement over doing multiple operations for every individual element of the matrix repeated for each column. Each row of the matrix has to be broken down into multiple vectors since the vector size is restricted in order to fit into registers. We used the 128 bit vector type which contains four 32 bit signed integers for our implementation. For a 10x10 matrix it is therefore necessary to use 3 vectors for every row. The following diagram, **Figure 3**, shows how the neon vector operations reduce multiple instructions into 1 for scalar multiplication.

Figure 3: Multiply vector by scalar

The same principle is used for subtracting the values using vector addition. In order to subtract using addition, the scalar that’s applied to the row is simply inverted beforehand. A diagram depicting vector addition is also included in **Figure 4**:

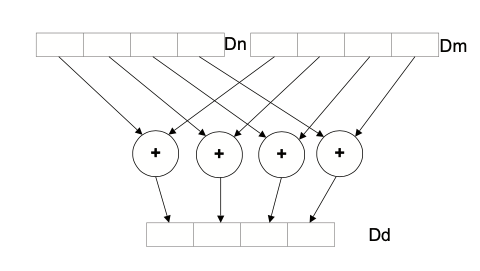


Figure 4: Vector addition

Our implementation keeps the matrix in an array and loads the data into vectors each time we want to do arithmetic on a row, which means there’s significant overhead. This overhead could be avoided by always storing the matrix in vectors, but this negatively affects the performance of other parts of the program, such as getting the largest value in a column for pivoting. That approach still introduces overhead in the other parts of the algorithm where single elements of each row are needed and furthermore, using vectors extensively across the source code is more complex to implement than only using them in the specific place where they are most needed. The following code in **Figure 5**, shows the vector operations performed for a single row.

Figure 5: Vector operations performed for a single row

|  |
| --- |
| int\_type ratio = second/first;  // load 1 row into 3 vectors  int32x4\_t v1 = vld1q\_s32(augmented[i]);  int32x4\_t v2 = vld1q\_s32(augmented[i]+4);  int32x4\_t v2 = vld1q\_s32(augmented[i]+8);  // scale the row to eliminate  v1 = vaddq\_s32(v1, vmulq\_n\_s32(v1, -ratio));  v2 = vaddq\_s32(v2, vmulq\_n\_s32(v2, -ratio));  v3 = vaddq\_s32(v2, vmulq\_n\_s32(v2, -ratio));  // store vectors back into row  vst1q\_s32(augmented[i], v1);  vst1q\_s32(augmented[i]+4, v2);  vst1q\_s32(augmented[i]+8, v2); |

## 3.3 Hardware assisted solution

In order to optimize further we developed a hardware assisted solution. We make use of a hardware component that eliminates the column elements other than the diagonal from a matrix. We then defined an assembly instruction that drives this hardware component in order to make use of it in our source code. The performance of the hardware component was tested based on an estimation of how long it would take to perform the elimination of 1 column. We estimate that each element would take 4 cycles to perform a load, multiply, add and store. Then for a 10x10 matrix this comes out to an estimated 400 cycles to eliminate 1 column.

The theoretical hardware instruction for vanishing the column elements is defined as follows.

Given the address of the target matrix and the index of the target column, it will eliminate all the elements in the column other than the element on the diagonal. This is performed by subtracting a scaled version of the unchanged row from each of the other rows in order to set them to zero. This process also affects the elements in other columns in accordance with the principles of performing row operations for Gaussian elimination. This instruction replaces the 2 inner loops of the eliminate function, as it is now only necessary to loop through each column and call the hardware instruction. The following code in **Figure 6**, shows the function in c code that would be called to use the hardware assist.

|  |
| --- |
| void hardware\_vanish\_column(int\_type[size][size\*2] a, int column)  {  asm (  "cvan %0, %1"  : "=r" (a)  : "r" (column)  );  } |

Figure 6: Hardware assist c code function

# 4 Discussion

The following section contains our numerical results and analysis for the performance testing of each of our optimizations.

## 4.1 Results

After completing both the pure software solution and the hardware solution, we compared the solutions by testing their performance. We compiled the program using the lab machine and ran it on the Fedora 29 minimal arm machine Both solutions ran for 100,000 repetitions in order to obtain an accurate execution time. By dividing the resulting execution time by the number of repetitions, we were able to get the result for one program execution. For the software solution, the tests were run for both the unoptimized and optimized code.

As expected, the optimized code executed 20% faster than the unoptimized code. To optimize the software solution, we added various optimization techniques and measured the execution times. Using CPU registers to store frequently used variables brough the execution time for 100,000 repetitions down from 77.27 seconds to 62.84 seconds, slighting increasing the performance. Moreover, by adding loop fission to achieve better locality of reference, we were able to get the execution time down to 62.07 seconds. The following show, in **table 1**, shows the results of our performance tests including the improvement percentage as compared to an unoptimized solution.

|  |  |  |
| --- | --- | --- |
| Code version | Average execution time (ms) | Improvement |
| Unoptimized | 0.772 | - |
| Registers | 0.628 | 18.67% |
| Loop Fission | 0.621 | 19.67% |
| Vector operations | 0.426 | 44.89% |
| Hardware assisted | 0.103 | 86.71% |

Table 1: Performance test results

## 4.2 Performance analysis

The following is further discussion on the specific performance results for each of the implemented optimizations.

### 4.2.1 CPU Registers

The register keyword was used to reduce traffic to memory thereby saving cycles in the algorithm used to find the maximum element in a column. This functionality is essential to the algorithm since it finds the pivot row that is necessary to guarantee the stability of the algorithm. The function is called once for every column, and it has to finish before the swapping and elimination steps can be performed, therefore it is a bottleneck in the overall algorithm.

The use of registers is well suited to optimizing this code section as it makes frequent use of a small number of local variables. Namely the variable used to store the maximum which gets compared to every element in the column, as well as the index variable which needs to be updated whenever a new maximum is found in order to return the index to be able to swap the rows. Because of the high frequency of access of these variables, the simple optimization of using the register keyword allows for a sizable increase in performance of about 20%.

### 4.2.2 Loop Fission

The next optimization of interest is the use of loop fission. This was used in the row swapping algorithm, where one loop iterates through the 2 rows and swaps each element using a temporary variable. Loop fission in this case involves splitting into 3 loops. In theory this could cause more load and store operations to occur due to needing more variables. In practice this optimization had a large impact of a 20% improvement over the unoptimized version. The main cause of this is that the unoptimized version created particularly inefficient assembly code for the swap function. For one thing, the unoptimized version does not make use of CPU registers for the temp variable. The evidence for this is that we found that manually requiring this with the register keyword in this instance had a similar performance boost to the loop fission. The other issue with the assembly in the unoptimized version is the greater number of branch instructions and memory access. The assembly in the loop fission version is much shorter and only uses 1 branching operation. We believe this is because the simplified loops in the loop fission version are able to be automatically unrolled by the compiler creating much more efficient code. For comparison, the body of the loop fission assembly subroutine uses 2 labels while the unoptimized version uses 6.

### 4.2.3 Vector operations

The use of vector operations gave the largest performance boost of all the software optimizations we employed for this project, with an improvement of about 45% over the unoptimized version. This was expected due to the powerful capability of the neon instruction set to manipulate multiple matrix elements using a single instruction. Because our implementation made use of 128 bit vectors storing 4 matrix elements each, one would extrapolate that the matrix inversion happens 4 times as fast as otherwise. However this was not the case, as is shown by our experimental results.

This phenomenon can be explained by 2 important factors in the use of vector operations for this application. Firstly, the matrix elements are being stored in an array for the majority of the algorithm, they are only stored into vectors at the time the row operation occurs, and are subsequently removed from the vectors before continuing on to the next row. This introduces a significant overhead, which is further increased by the fact that each row must be stored in multiple vectors due to their size limitation. And because our implementation handles matrices of sizes up to 10x10, the rows are stored using 3 vectors, the third of which only stores 2 elements which further reduces the theoretical efficiency gain of 4 times.

The other major factor is inherent to the Gaussian elimination algorithm. As the algorithm progresses the matrix is diagonalized, this means that more and more of the elements in the matrix are set to 0. Arithmetic operations performed on these 0 elements never have any effect and so storing these extra zeroes into vectors does not provide any benefit. For this reason the vector operations only give their maximal performance boost in the first half the algorithm before losing efficiency in the later stages when many elements have been zeroed out.

### 4.2.4 Hardware assist

For the performance testing of the solution that makes use of the theoretical new hardware component, we had to simulate the operations of the hardware construction based on the provided estimate for the instruction’s runtime. We simulated this runtime using a simple spin loop in place of the instruction. We measured a very high performance boost since by design the hardware component does the specific kind of calculation necessary for this algorithm in a very short period of time. For this reason the performance increase of upwards of 80% is more or less expected. We expect that building this hardware component in practice would be comparatively expensive due to its specialized nature and the high number of operations that it carries out. A hardware assisted matrix inversion of this kind would probably not be necessary in an embedded context for matrices of this type, but the performance gains would be highly beneficial for applications working with very large matrices.

## 4.3 General discussion

Throughout the work of the project, various difficulties arose. For the software solution, a few matrices weren’t providing the correct inversion, resulting in only a few rows with correct values and the remaining rows being zeros. The issue ended up being when a column got eliminated the singular remaining value was still occasionally getting chosen as a pivot, resulting in swapping the row and providing an inaccurate answer. Additionally, originally we were approaching the algorithm without dividing the rows, only multiplying and subtracting. This produced a correct result for small matrices but for larger matrices, the values would get too large and cause overflow. To fix this issue, we added division but to avoid any fixed point rounding, we increased the scale factor for the identity matrix to minimize the rounding.

# 5 Conclusion

The purpose of this project was to compare a pure software solution and a hardware assisted solution of inverting a matrix using Gauss-Jordan Elimination with row interchanges. The matrices were restricted to no larger than a 10x10 matrix with a 12-bit word length input. Moreover, to ensure our input data didn’t affect the computer answer, we calculated the condition number for each matrix and included an ill-conditioned matrix to observe the effects of how the error in the input data affected the result.

The hardware assisted solution used fewer operations resulting in an execution time significantly faster than the pure software solution. Our group learned the importance of optimizing a software based solution through various techniques learned in class. Through these techniques we were able to bring the execution time down and improve the overall performance of the program. To increase the performance of the software solution, future work could be done to optimize the code more by exploring new optimization techniques. Additionally, increasing the restricted max size of the matrix would allow us to test the strength of our algorithm under more strenuous conditions. The overall outcome of the project demonstrated the benefits of optimizing code and the difference between pure software and hardware assisted programs.

# 

# 6 References

[1] M. Sima, "Lesson 111 - Matrix Inversion", 2019.

[2] M. Taboga, “Gauss Jordan Elimination algorithm,” *Gauss Jordan elimination*. [Online]. Available: https://www.statlect.com/matrix-algebra/Gauss-Jordan-elimination. [Accessed: 09-Aug-2022].

[3] M. Sima, “Lesson 5: Software Optimization Techniques III”, 2019

[4] M. Sima, “Lesson 3: Software Optimization Techniques I”, 2019

[5] Arm, “NEON Programmer’s Guide”, 2013

# 

# 7 Appendices

## Appendix A - General Inversion C code

#include <stdio.h>

#include <stdint.h>

#include <time.h>

#include <math.h>

#include <stdlib.h>

#include <assert.h>

#include <unistd.h>

#include "inverse.h"

#define SHIFT\_AMOUNT 20

#define SHIFT\_MASK ((1 << SHIFT\_AMOUNT) -1)

int\_type augmented[size][size\*2];

float float\_matrix[size][size];

int\_type augmented[size][size\*2];

// prints the augmented matrix

void print(int\_type a[size][size\*2])

{

printf("{\n");

for(int i = 0; i < size; i++)

{

for(int j = 0; j < size\*2; j++)

{

printf("% 4lld, ", a[i][j]);

}

printf("\n");

}

printf("}\n");

}

// augments the square input matrix with the identity matrix and scales values for fixed point arithmetic

void augment()

{

for(int i = 0; i < size; i++)

{

for(int j = 0; j < size; j++)

{

// copying input matrix

augmented[i][j] = matrix[i][j] << SHIFT\_AMOUNT;

// putting in identity matrix

augmented[i][j+size] = ((i == j) << SHIFT\_AMOUNT);

}

}

}

// swaps 2 rows for use in row interchange

void swap(int row1, int row2)

{

int\_type temp[size\*2];

for(int i = 0; i < size\*2; i++)

{

temp[i] = augmented[row1][i];

}

for(int i = 0; i < size\*2; i++)

{

augmented[row1][i] = augmented[row2][i];

}

for(int i = 0; i < size\*2; i++)

{

augmented[row2][i] = temp[i];

}

}

// returns the index of the largest element in a column for pivoting

int max\_column(int col)

{

register int\_type max = augmented[0][col];

register int index = 0;

for(int i = 1; i < size; i++)

{

int\_type current = augmented[i][col];

if(current < 0)

{

current \*= -1; // taking absolute value incase the max is negative

}

if(current> max)

{

max = current;

index = i;

}

}

return index;

}

// performs gaussian elimination with full pivoting

void eliminate()

{

int\_type first;

int\_type second;

for(int k = 0; k < size; k++)

{

// finding the pivot row

int pivot\_index = max\_column(k);

if(k < pivot\_index)

{

swap(k, pivot\_index);

}

first = augmented[k][k];

for(int i = 0; i < size; i++)

{

second = augmented[i][k];

if(second == 0)

{

continue; // skip rows where its already 0

}

if(i != k)

{

for(int j = 0; j < size\*2; j++)

{

// this step does the elimination and subtracts across the row

augmented[i][j] -= second\*augmented[k][j] / first;

}

}

}

}

}

// The final matrix is printed as floating point for verification purposes

void print\_fractional()

{

for(int i = 0; i < size; i++)

{

double normal = augmented[i][i]; // normalizing the integer values to get back floating point

for(int j = 0; j < size; j++)

{

float\_matrix[i][j] = augmented[i][j+size] / normal;

}

}

// now printing

printf("{\n");

for(int i = 0; i < size; i++)

{

for(int j = 0; j < size; j++)

{

printf("%f, ", float\_matrix[i][j]);

}

printf("\n");

}

printf("}\n");

}

// prints out preprogramed expected matrix for testing

void print\_expected()

{

printf("{\n");

for(int i = 0; i < size; i++)

{

for(int j = 0; j < size; j++)

{

printf("%f, ", inv\_matrix[i][j]);

}

printf("\n");

}

printf("}\n");

}

void end\_timer(clock\_t start\_time)

{

clock\_t end\_time = clock();

double time\_spent = (double)(end\_time - start\_time) / CLOCKS\_PER\_SEC;

printf("%f seconds to execute\n", time\_spent);

}

int main(int argc, char \*argv[])

{

augment();

// start the timer

clock\_t start\_time = clock();

// do gaussian elimination

eliminate();

// end timer and print time elapsed

end\_timer(start\_time);

// print out results for verification

printf("Fractional result:\n");

print\_fractional();

printf("Expected result:\n");

print\_expected();

}

## Appendix B - Optimization code sections

B.1 Vector operation version of eliminate function

void eliminate() {

int\_type first;

int\_type second;

for(int k = 0; k < size; k++) {

int pivot\_index = max\_column(k);

if(k < pivot\_index) {

swap(k, pivot\_index);

}

first = augmented[k][k];

for(int i = 0; i < size; i++) {

second = augmented[i][k];

if(second == 0) continue; // skip rows where its already 0

if(i != k) {

int\_type ratio = second/first;

// load 1 row into 3 vectors

int32x4\_t v1 = vld1q\_s32(augmented[i]);

int32x4\_t v2 = vld1q\_s32(augmented[i]+4);

int32x4\_t v2 = vld1q\_s32(augmented[i]+8);

// scale the row to eliminate

v1 = vaddq\_s32(v1, vmulq\_n\_s32(v1, -ratio));

v2 = vaddq\_s32(v2, vmulq\_n\_s32(v2, -ratio));

v3 = vaddq\_s32(v2, vmulq\_n\_s32(v2, -ratio));

// store vectors back into row

vst1q\_s32(augmented[i], v1);

vst1q\_s32(augmented[i]+4, v2);

vst1q\_s32(augmented[i]+8, v2);

}

}

}

}