SENG 440 - Embedded Systems Matrix Inversion

Amaan Makhani V00883520
Evan Roubekas V00891470

Summer 2021, Team 15

Outline

- Project Specification
- Background Information
- Software Solution
- Optimization to software
- Hardware Inlining
- Simulations and Cycles
- Conclusion and Observations

Project Specifications

- Matrix inversion
- 13-bit signed integers
- 10X10 matrix
- A test bench containing:
  - A well-conditioned matrix
  - A ill-conditioned matrix
- Using Gauss-Jordan algorithm with pivoting and integer arithmetic
- A pure-software solution, and an optimized software solution
- Hardware support for vanishing the column elements

Background Information

- Condition number, $\kappa = ||A|| \cdot ||A^{-1}||$
- Where $||A||$ and $||A^{-1}||$ is the norm of the matrix
- The norm is calculated as follows:

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

- A well conditioned matrix is when κ is small relative to one
  - A small relative change -> small relative error in the inverse
- A large condition number
  - A small relative change -> large relative error in the inverse.

Background Information

- Gauss-Jordan elimination
  - A matrix is converted into a system of equations
  - All matrix elements vanished except one element per column
  - Manipulates the identity matrix at the same time as manipulating the matrix
  - The identity matrix is one where the one’s are located on the diagonal
- Problems:
  - Increase in cache misses
  - Can fail when an attempted division by zero occurs

Software Solution

Gauss-Jordan elimination with pivoting

- The largest element along the column is declared the pivot/ value on the diagonal
- More calls to memory

Process:

1. Find 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

   The row element - (column element to eliminate * corresponding row element from the row the one was produced in)

This is done for each column

Software Solution

2)

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

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

3)

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

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

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

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

Test Bench

- Ill conditioned matrix with condition number of 5748
  - 12 extra bits needed to estimate inverse

```
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}
};
```

- Well conditioned matrix with condition number of 15
  - 4 extra bits needed to estimate inverse

```
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}
};
```

Optimizations to Software

Techniques Used:

- Remove functions calls / inlining
- Register / unsigned / short int
- Loop Unrolling
- Operator Strength Reduction

Before

```
> int norm(int matrix[10][10]){ ···
  }

> void swapRows(int a, int b, int matrix[10][10]) { ···
  }
```

```
int inverseNorm = norm(matrix);
int conditionNum = matrixNorm * inverseNorm;
int numBits = 0;
```

```
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);
        }
    }
}
```

Optimizations to Software

After

```
static inline int norm(int matrix[10][10]){ ···
}

int main()
{
```

```
short register unsigned int inverseNorm = norm(matrix);
short register unsigned int conditionNum = matrixNorm * inverseNorm;
short register unsigned int numBits = 0;
```

```
int multiplier;
for(m = 0; m < 10; m++) {
    if (m != i) {
        multiplier = matrix[m][i]/4096;
        for(j = 0; j < 10; j+=2) {
            matrix[m][j] = matrix[m][j] - (multiplier * matrix[i][j]);
            inverse[m][j] = inverse[m][j] - (multiplier * inverse[i][j]);

            matrix[m][j+1] = matrix[m][j+1] - (multiplier * matrix[i][j+1]);
            inverse[m][j+1] = inverse[m][j+1] - (multiplier * inverse[i][j+1]);
        }
    }
}
```

Hardware Inlining

- Requirement to use hardware support to aid in vanishing column element operation
- Use of assembly sub command to achieve the subtraction

Pure Software

```
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);
        }
    }
}
```

Hardware Inlining

```
for(int m = 0; m < 10; m++) {
    if (m != i) {
        int multiplier = matrix[m][i];
        for(int j = 0; j < 10; j++) {
            // Hardware inlining for vanishing column elements
            __asm__ ("sub\t%0, %1" : "=r" (matrix[m][j]) : "r" (multiplier * matrix[i][j]/4096));
            __asm__ ("sub\t%0, %1" : "=r" (inverse[m][j]) : "r" (multiplier * inverse[i][j]/4096));
        }
    }
}
```

Simulations and Cycles / Instructions

Simulation	Well Conditioned Matrix (Assembly Instructions)	Ill Conditioned Matrix (Assembly Instructions)
Software Solution not Optimized	646	646
Optimized Software Solution	620	585
Software Solution not Optimized with Hardware Inlining	632	632
Optimized Software Solution with Hardware Inlining	797	868

Conclusion and Observations

- Matrix Inversion is an important process that has many real world applications
- Even with a good algorithm and large scale factor some matrices might have condition numbers so large that a unreasonable amount of extra bits are needed to estimate its inverse
- Using software optimization techniques, we optimized the Gauss-Jordan algorithm with pivoting, which resulted in a decrease of ~44 lines
- The proposed hardware solution provides an improvement when not combined with software optimization

Questions

We are happy to answer any questions.

Thank you for listening!