ELL 782: Computer Architecture, Assignment-1

2024EET2368 - ANIMESH LOHAR

September 1, 2024

1 Introduction

This RISC-V assembly program performs matrix inversion using the Gaussian elimination method. The program is designed to handle a 5x5 floating-point matrix and includes several key modules for matrix operations, such as copying matrices, swapping rows, performing Gaussian elimination, checking for matrix invertibility, and verifying the computed inverse.

Data Initialization

The following matrices and variables are initialized:

- **A**: A 5x5 matrix represented as a flattened array of 25 floating-point numbers.
- **I**: A 5x5 identity matrix, also represented as a flattened array.
- **B**: A 5x5 matrix to hold the modified identity matrix during the Gaussian elimination process.
- **C**: A 5x5 matrix to hold a copy of matrix **A**.
- **D**: An additional 5x5 matrix reserved for possible intermediate results.
- **n**: The size of the matrix, which is set to 5.
- **fmt**: A format string for printing floating-point numbers.
- **no_solution_string**: A string that will be printed if the matrix inversion fails.

Algorithm

The algorithm proceeds as follows:

1. **Copy the Identity Matrix to B**:
$$B[i] = I[i] \quad \text{for } i = 0 \text{ to } 24$$
2. **Copy Matrix A to C**:
$$C[i] = A[i] \quad \text{for } i = 0 \text{ to } 24$$
3. **Perform Row Swapping**:
$$\text{Swap rows in matrix } C \text{ using matrix } B$$
4. **Apply Gaussian Elimination**:
$$\text{Transform matrix } C \text{ to row echelon form and apply the same operations to matrix } B$$
5. **Check for Solution**:
    - If matrix **C** is singular, print "Inverse of the matrix does not exist."
    - Otherwise, the matrix **B** will contain the inverse of matrix **A**.

Pseudo Code

```
START
    # Initialize matrices and variables
    Load identity matrix I into B
    Load matrix A into C

    # Copy matrix I into B
    FOR i FROM 0 to 24 DO
        B[i] = I[i]
    END FOR

    # Copy matrix A into C
    FOR i FROM 0 to 24 DO
        C[i] = A[i]
    END FOR

    # Perform row swapping on matrix C using matrix B
    CALL row_swapping(C, B)

    # Perform Gaussian elimination on matrix C
    # to find the inverse in matrix B
    CALL gauss_elimination(C, B)
END
```

Finding Pivot and swapping Assembly Code Explanation

Copy Loop

The `copy_loop` section copies elements from matrix `I` to matrix `B`:

- `beq t2, t3, end_copy`: Exit the loop if all elements are copied.
- `flw ft0, 0(t0)`: Load a floating-point element from `I`.
- `fsw ft0, 0(t1)`: Store the floating-point element into `B`.
- `addi t0, t0, 4`: Move to the next element in `I`.
- `addi t1, t1, 4`: Move to the next element in `B`.
- `addi t2, t2, 1`: Increment the index.
- `j copy_loop`: Jump back to continue copying the next element.

Row Swapping

The `row_swapping` section swaps rows in a matrix if the first element of a row is zero:

- `li t2, 0`: Load the bit pattern for `0.0`.
- `flw f1, 0(t0)`: Load the first element of the row.
- `fcvt.w.s t3, f1`: Convert the floating-point value to an integer.
- `beq t2, t3, non_zero_row`: Check if the first element is zero.
- `jalr x0, 0(ra)`: Return if no swap is needed.

Pseudo Code

```
copy_loop:
    WHILE (index != total_elements):
        Load element from matrix I
        Store element into matrix B
        Move to the next element in I and B
        Increment index
    END WHILE
    EXIT

row_swapping:
    IF (first element of row is 0):
        FOR each row:
            IF (non-zero element found):
                Swap rows
                EXIT
            END IF
        END FOR
        PRINT "No inverse possible"
    END IF
    EXIT

swapping:
    FOR each column:
        Swap elements between the two rows
    END FOR
    EXIT
```

Explanation of the Converting into Upper Triangular Matrix Assembly Code

The assembly code is structured into several key components:

- **Initialization**: Load the matrix size into a register and set up the initial row and column indices.
- **Most Outer Loop**: This loop iterates over each column of the matrix.
- **Outer Loop**: This loop iterates over the rows of the current column, performing the necessary calculations.
- **Inner Loop**: This loop updates the matrix elements by subtracting the appropriate scaled values to achieve an upper triangular form.

Pseudo Code

The following pseudo code summarizes the Gaussian elimination process implemented in the RISC-V assembly code:

```
Initialize matrix size n
Set current_row to 1
Set current_column to 0

# Convert matrix C to upper triangular form
while current_column < n do
    while current_row < n do
        Set inner_row to current_column + 1
```

```
        Calculate offset = 4 * n * current_row
        Load base addresses of matrices C and B
        Load C[current_column][current_column] into factor
        Load C[current_row][current_column] into element
        Calculate scaling_factor = element / factor

        # Inner loop: Update rows based on scaling factor
        while inner_row < n do
            Load elements from matrices C and B
            Update C and B using the scaling factor
            Move to the next element and increment inner_row
        end while

        Increment current_row
    end while

    Increment current_column
end while
```

Pseudo Code for Checking Inverse exist

```
CHECK_INVERSE_EXIST
    Load address of matrix B into t1
    Load matrix size n into a2
    Initialize loop counter a3 to 1
    Set a4 to 4 (bytes per element)
    Compute total number of bytes in matrix (n * 4) and store in a5
    Initialize index a7 to 0
    Initialize floating-point register f1 to 1.0
LOOP checking_loop
        If a3 >= n, exit loop
        Compute byte offset for current row and element
        Update address t1 to point to current element
        Load matrix element into floating-point register f2
        Multiply f1 by f2
        Increment loop counter a3
        Increment index a7
        Jump back to start of loop

    END_LOOP
    Initialize t4 to 0
    Convert t4 to floating-point and store in f2
    Compare f1 (product of all elements) with f2 (0)
    If they are equal, jump to no_inverse (matrix is not invertible)
    Else, jump to converting_identity (matrix is invertible)
```

Converting Matrix to Identity Matrix Form

- **Load Matrix Size**: Load the matrix size n.
- **Load Matrix Addresses**: Load the base addresses of matrices A and I.
- **Outer Loop**: Iterate over rows of matrix A.
- **Check and Scale Diagonal Element**: If the diagonal element is not 1.0, scale the row to make it 1.0.
- **Inner Loop**: Apply the scaling to each element in the row.
- **Copy Matrix**: Copy the scaled matrix I back to matrix A.

Pseudo Code for Converting to Identity Matrix

```
function converting_identity:
    load matrix size n into a0
    load base address of A into t0
    load base address of I into t1

    for i from 0 to n-1:
        calculate offset for diagonal element
        load diagonal element A[i][i]
        if A[i][i] is not 1.0:
            scale row to make diagonal element 1
            for j from 0 to n-1:
                scale element A[i][j]
                store result in I[i][j]

    copy matrix I back to A
```

Checking the Inverse

- **Matrix Multiplication**: Multiply matrices C and B to check if B is the inverse of A.
- **Print Matrices**: Print matrices B and D to display the results.

Pseudo Code for Checking the Inverse

```
function check_inverse:
    load base address of matrices A, B, and D
    load matrix size n

    for i from 0 to n-1:
        for j from 0 to n-1:
            initialize sum for C[i][j]
            for k from 0 to n-1:
                compute partial product A[i][k] * B[k][j]
                accumulate to sum
            store sum in C[i][j]

    print matrix B
    print matrix D
```

2 Code Explanation

2.1 Matrix Copying

The code begins by copying the identity matrix I into matrix B and the original matrix A into matrix C. This is accomplished using the copy$_{l}ooproutine, whichiteratesovereachelementandperformsfloating-pointloadsandstores.$

2.2 Row Swapping

The row$_{s}wappingroutineensuresthateachrowhasanon-zeroleadingcoefficient.Itscansthroughtherowstofindanon-zerorowandswapsitwiththecurrentrowifnecessary.ThisiscriticalfortheGaussianeliminationprocess.$

2.3 Gaussian Elimination

The gauss$_{e}liminationroutineperformsthecorematrixoperations.Itconvertsthematrixintoanuppertriangularformby$

2.4 Check for Inverse

After Gaussian elimination, the code checks if the matrix is invertible. It verifies if the matrix has a non-zero determinant by multiplying the diagonal elements. If the determinant is zero, it outputs that the matrix does not have an inverse.

2.5 Converting to Identity Matrix

If the matrix is invertible, the code converts the upper triangular matrix into the identity matrix by scaling rows so that the diagonal elements are 1.0.

2.6 Matrix Multiplication

The check$_inverseroutineperformsmatrixmultiplicationofmatrixC$ (the original matrix) and matrix B (the inverse matrix). The result is stored in matrix D.

2.7 Printing Matrices

The code includes routines for printing matrices B and D. This is done by iterating over the matrix elements, converting them to integers, and printing them using system calls.

3 What I have done extra

I have done the converting upper triangular matrix into minimum size of code.

4 Conclusion

The RISC-V assembly code efficiently handles matrix inversion and multiplication for 5×5 matrices. The detailed explanation covers matrix copying, row swapping, Gaussian elimination, and matrix operations, ensuring a comprehensive understanding of the implementation.