[Matrix multiplication animation - YouTube](https://www.youtube.com/watch?v=1hf_cHNbgCk)

[Matrix Multiplication: How to Multiply Two Matrices Together. Step by step visual animation and interactive practice problems](https://www.mathwarehouse.com/algebra/matrix/multiply-matrix.php)

[Matrix Multiplication Animated Example – GeoGebra](https://www.geogebra.org/m/ETHXK756)

[Deep Learning from Scratch in Modern C++: Convolutions | Towards AI](https://towardsai.net/p/l/deep-learning-from-scratch-in-modern-c-convolutions)

Matrix multiplication is a fundamental operation in various fields, including machine learning, computer vision, and speech processing.

In machine learning, it is crucial for operations such as transforming data, calculating weights in neural networks, and performing linear transformations in algorithms like Principal Component Analysis (PCA).

In computer vision, matrix multiplication is used for image transformations, such as scaling, rotating, and translating images, as well as in convolution operations for feature extraction in convolutional neural networks (CNNs).

In speech processing, it plays a role in filtering and transforming audio signals, as well as in the implementation of algorithms for speech recognition and synthesis.

Overall, matrix multiplication enables efficient computation and manipulation of data, making it indispensable in these advanced technological domains.

Code snippet

# Eazy68k Assembly (Matrix Multiplication 3x3)

MOVE.L #matrix1,A0 ; Load address of matrix1 into A0

MOVE.L #matrix2,A1 ; Load address of matrix2 into A1

MOVE.L #result,A2 ; Load address of result matrix into A2

; Calculate result[0][0]

MOVE.L 0(A0),D0 ; D0 = matrix1[0][0]

MOVE.L 0(A1),D1 ; D1 = matrix2[0][0]

MULS D1,D0 ; D0 = matrix1[0][0] \* matrix2[0][0]

MOVE.L 4(A0),D2 ; D2 = matrix1[0][1]

MOVE.L 12(A1),D3 ; D3 = matrix2[1][0]

MULS D3,D2 ; D2 = matrix1[0][1] \* matrix2[1][0]

ADD.L D2,D0 ; D0 = D0 + D2

MOVE.L 8(A0),D4 ; D4 = matrix1[0][2]

MOVE.L 24(A1),D5 ; D5 = matrix2[2][0]

MULS D5,D4 ; D4 = matrix1[0][2] \* matrix2[2][0]

ADD.L D4,D0 ; D0 = D0 + D4

MOVE.L D0,0(A2) ; result[0][0] = D0

; Calculate result[0][1]

MOVE.L 0(A0),D0 ; D0 = matrix1[0][0]

MOVE.L 4(A1),D1 ; D1 = matrix2[0][1]

MULS D1,D0 ; D0 = matrix1[0][0] \* matrix2[0][1]

MOVE.L 4(A0),D2 ; D2 = matrix1[0][1]

MOVE.L 16(A1),D3 ; D3 = matrix2[1][1]

MULS D3,D2 ; D2 = matrix1[0][1] \* matrix2[1][1]

ADD.L D2,D0 ; D0 = D0 + D2

MOVE.L 8(A0),D4 ; D4 = matrix1[0][2]

MOVE.L 28(A1),D5 ; D5 = matrix2[2][1]

MULS D5,D4 ; D4 = matrix1[0][2] \* matrix2[2][1]

ADD.L D4,D0 ; D0 = D0 + D4

MOVE.L D0,4(A2) ; result[0][1] = D0

; Calculate result[0][2]

MOVE.L 0(A0),D0 ; D0 = matrix1[0][0]

MOVE.L 8(A1),D1 ; D1 = matrix2[0][2]

MULS D1,D0 ; D0 = matrix1[0][0] \* matrix2[0][2]

MOVE.L 4(A0),D2 ; D2 = matrix1[0][1]

MOVE.L 20(A1),D3 ; D3 = matrix2[1][2]

MULS D3,D2 ; D2 = matrix1[0][1] \* matrix2[1][2]

ADD.L D2,D0 ; D0 = D0 + D2

MOVE.L 8(A0),D4 ; D4 = matrix1[0][2]

MOVE.L 32(A1),D5 ; D5 = matrix2[2][2]

MULS D5,D4 ; D4 = matrix1[0][2] \* matrix2[2][2]

ADD.L D4,D0 ; D0 = D0 + D4

MOVE.L D0,8(A2) ; result[0][2] = D0

; Calculate result[1][0] ... (Similar pattern, omitted for brevity)

; ...

; Calculate result[2][2] ... (Similar pattern, omitted for brevity)

; ...rest of the calculation...

; Exit

TRAP #15 ; Exit to the operating system

matrix1:

DC.L 1, 2, 3

DC.L 4, 5, 6

DC.L 7, 8, 9

matrix2:

DC.L 9, 8, 7

DC.L 6, 5, 4

DC.L 3, 2, 1

result:

DS.L 9 ; Allocate space for the 3x3 result matrix

# RISC-V Assembly (Matrix Multiplication 3x3)

.data

matrix1: .word 1, 2, 3, 4, 5, 6, 7, 8, 9

matrix2: .word 9, 8, 7, 6, 5, 4, 3, 2, 1

result: .space 36

.text

.globl \_start

\_start:

la t0, matrix1 # Load address of matrix1 into t0

la t1, matrix2 # Load address of matrix2 into t1

la t2, result # Load address of result matrix into t2

# Calculate result[0][0]

lw t3, 0(t0) # t3 = matrix1[0][0]

lw t4, 0(t1) # t4 = matrix2[0][0]

mul t5, t3, t4 # t5 = matrix1[0][0] \* matrix2[0][0]

lw t3, 4(t0) # t3 = matrix1[0][1]

lw t4, 12(t1) # t4 = matrix2[1][0]

mul t6, t3, t4 # t6 = matrix1[0][1] \* matrix2[1][0]

add t5, t5, t6 # t5 = t5 + t6

lw t3, 8(t0) # t3 = matrix1[0][2]

lw t4, 24(t1) # t4 = matrix2[2][0]

mul t6, t3, t4 # t6 = matrix1[0][2] \* matrix2[2][0]

add t5, t5, t6 # t5 = t5 + t6

sw t5, 0(t2) # result[0][0] = t5

# Calculate result[0][1]

lw t3, 0(t0) # t3 = matrix1[0][0]

lw t4, 4(t1) # t4 = matrix2[0][1]

mul t5, t3, t4 # t5 = matrix1[0][0] \* matrix2[0][1]

lw t3, 4(t0) # t3 = matrix1[0][1]

lw t4, 16(t1) # t4 = matrix2[1][1]

mul t6, t3, t4 # t6 = matrix1[0][1] \* matrix2[1][1]

add t5, t5, t6 # t5 = t5 + t6

lw t3, 8(t0) # t3 = matrix1[0][2]

lw t4, 28(t1) # t4 = matrix2[2][1]

mul t6, t3, t4 # t6 = matrix1[0][2] \* matrix2[2][1]

add t5, t5, t6 # t5 = t5 + t6

sw t5, 4(t2) # result[0][1] = t5

# Calculate result[0][2]

lw t3, 0(t0) # t3 = matrix1[0][0]

lw t4, 8(t1) # t4 = matrix2[0][2]

mul t5, t3, t4 # t5 = matrix1[0][0] \* matrix2[0][2]

lw t3, 4(t0) # t3 = matrix1[0][1]

lw t4, 20(t1) # t4 = matrix2[1][2]

mul t6, t3, t4 # t6 = matrix1[0][1] \* matrix2[1][2]

add t5, t5, t6 # t5 = t5 + t6

lw t3, 8(t0) # t3 = matrix1[0][2]

lw t4, 32(t1) # t4 = matrix2[2][2]

mul t6, t3, t4 # t6 = matrix1[0][2] \* matrix2[2][2]

add t5, t5, t6 # t5 = t5 + t6

sw t5, 8(t2) # result[0][2] = t5

# Calculate result[1][0] ... (Similar pattern, omitted for brevity)

# ...

# Calculate result[2][2] ... (Similar pattern, omitted for brevity)

# ...rest of the calculation...

# Exit program

li a7, 93

ecall

Explanation and Key Points:

* Unfolding: Both code examples fully "unfold" the matrix multiplication, meaning they explicitly calculate each element of the result matrix without using loops. This makes the code longer but easier to understand step-by-step.
* Addressing:
  + Eazy68k uses address registers (A0, A1, A2) and offsets to access matrix elements.
  + RISC-V uses la (load address) to get the base addresses and offsets for lw (load word) and sw (store word).
* Multiplication:
  + Eazy68k uses MULS (signed multiply).
  + RISC-V uses mul.
* Addition:
  + Both use ADD.L or add to accumulate the multiplication results.
* Storage:
  + Both use MOVE.L or sw to store the calculated result into the result matrix.
* Comments: Comments are provided beside each instruction to explain its purpose.
* Omitted Code: To avoid excessive length, the calculations for the remaining result matrix elements (result[1][0] to result[2][2]) follow the same pattern and are omitted.
* Data Sections: Both codes properly define the input matrices and allocate space for the result matrix in the data sections.
* Exit: Both codes use the correct method to terminate the program.