MATMUL

INDEX:

INDEX:

INDEX:

INDEX:

INDEX:

INDEX:

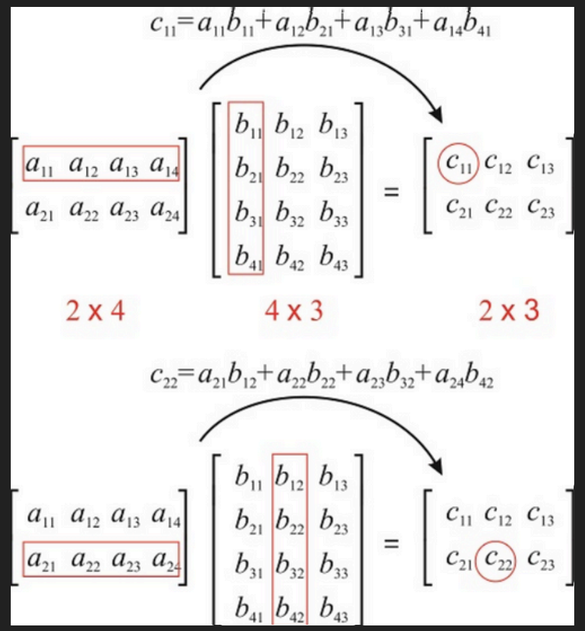
INDEX:

INDEX:

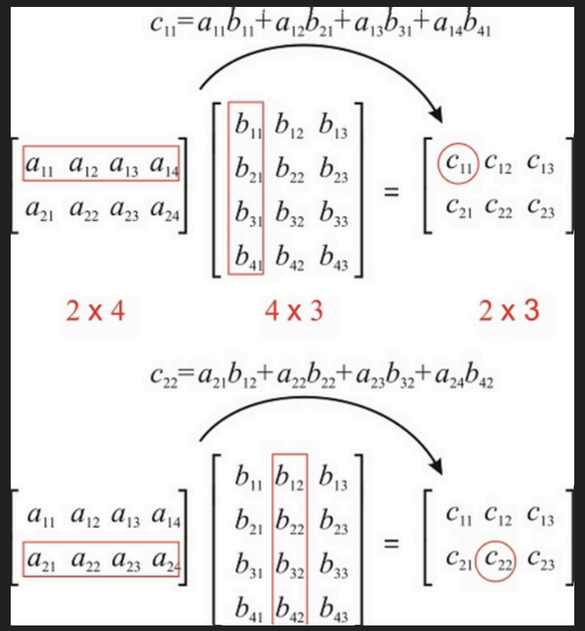
1. WHAT AND WHY OF MATRIX MULTIPLICATION

1. WHAT AND WHY OF MATRIX MULTIPLICATION

1. WHAT AND WHY OF MATRIX MULTIPLICATION

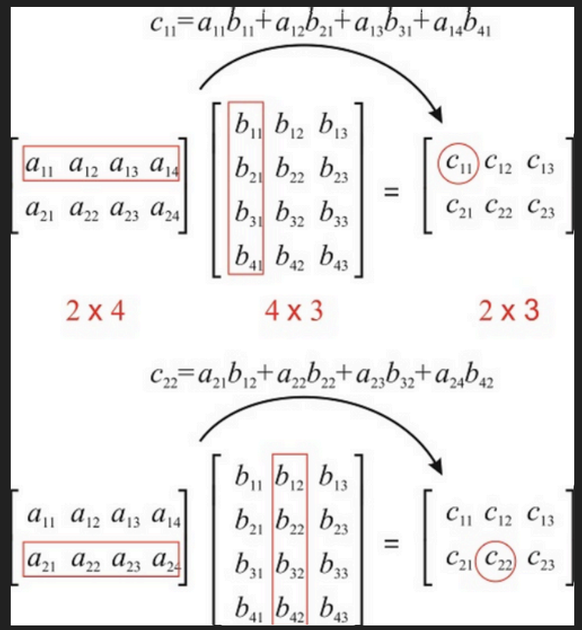


1. WHAT AND WHY OF MATRIX MULTIPLICATION



1. WHAT AND WHY OF MATRIX MULTIPLICATION

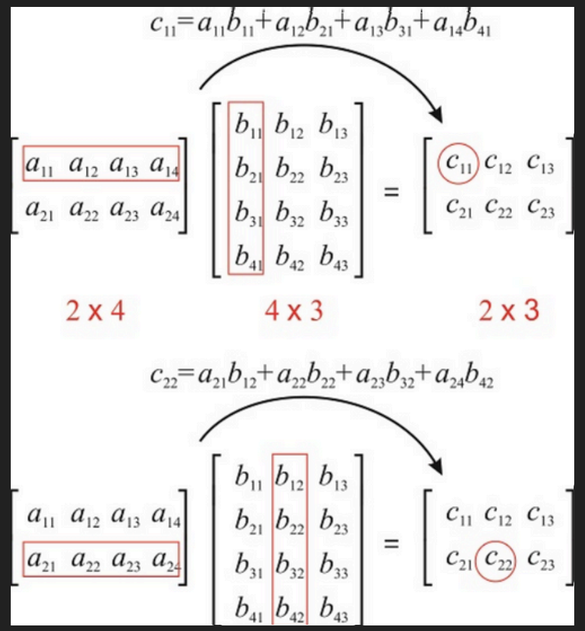
1. LINEAR ALGEBRA (LINEAR TRANSFORMATION, VECTOR SPACE, EQUATIONS AND CORE BUILDING BLOCK)



1. WHAT AND WHY OF MATRIX MULTIPLICATION

1. LINEAR ALGEBRA (LINEAR TRANSFORMATION, VECTOR SPACE, EQUATIONS AND CORE BUILDING BLOCK)

2. GRAPHICS AND VR (ROTATIONS, TRANSFORMATIONS, SCALING, VIEWPOINT, MODELLING)

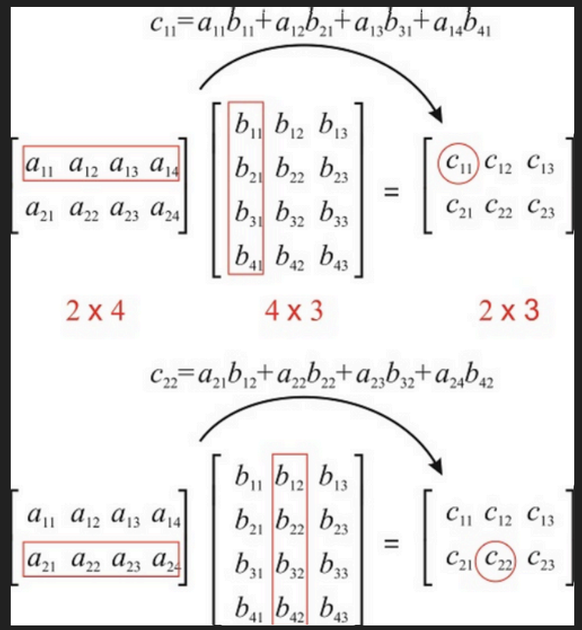


1. WHAT AND WHY OF MATRIX MULTIPLICATION

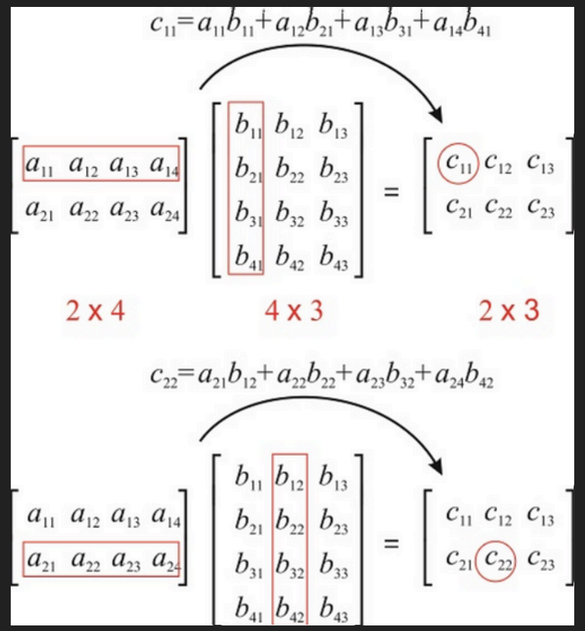
1. LINEAR ALGEBRA (LINEAR TRANSFORMATION, VECTOR SPACE, EQUATIONS AND CORE BUILDING BLOCK)

2. GRAPHICS AND VR (ROTATIONS, TRANSFORMATIONS, SCALING, VIEWPOINT, MODELLING)

3. ML & AI (ENCODING, MODELS, NEURAL NETWORK, IMAGE RECOGNITION, FNN, RNN, CNN)



1. WHAT AND WHY OF MATRIX MULTIPLICATION



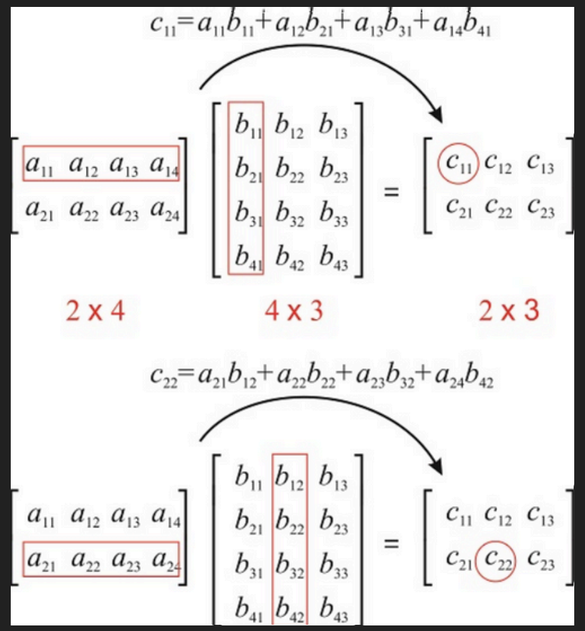
1. LINEAR ALGEBRA (LINEAR TRANSFORMATION, VECTOR SPACE, EQUATIONS AND CORE BUILDING BLOCK)

2. GRAPHICS AND VR (ROTATIONS, TRANSFORMATIONS, SCALING, VIEWPOINT, MODELLING)

3. ML & AI (ENCODING, MODELS, NEURAL NETWORK, IMAGE RECOGNITION, FNN, RNN, CNN)

4. ROBOTICS (SENSORY DATA PROCESSING, DYNAMIC MOVEMENT, NAVIGATION)

1. WHAT AND WHY OF MATRIX MULTIPLICATION



1. LINEAR ALGEBRA (LINEAR TRANSFORMATION, VECTOR SPACE, EQUATIONS AND CORE BUILDING BLOCK)

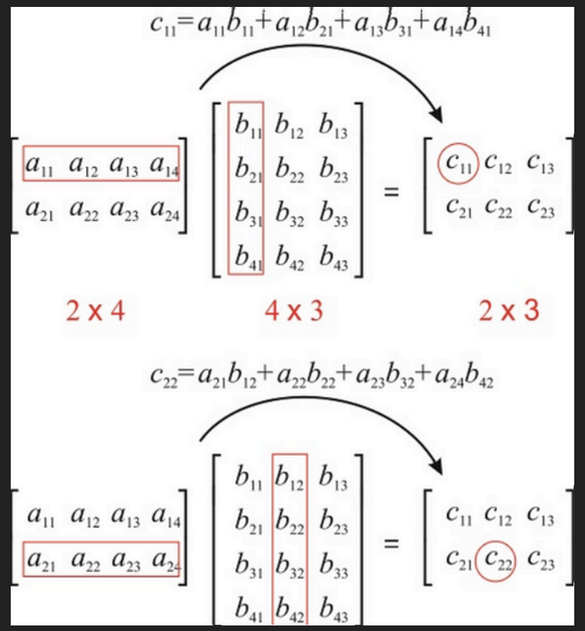
2. GRAPHICS AND VR (ROTATIONS, TRANSFORMATIONS, SCALING, VIEWPOINT, MODELLING)

3. ML & AI (ENCODING, MODELS, NEURAL NETWORK, IMAGE RECOGNITION, FNN, RNN, CNN)

4. ROBOTICS (SENSORY DATA PROCESSING, DYNAMIC MOVEMENT, NAVIGATION)

5. DATA ANALYTICS (PREPROCESSING, OPERATIONS, INTERACTIONS, REAL-TIME DECISION MAKING)

1. WHAT AND WHY OF MATRIX MULTIPLICATION



1. LINEAR ALGEBRA (LINEAR TRANSFORMATION, VECTOR SPACE, EQUATIONS AND CORE BUILDING BLOCK)

2. GRAPHICS AND VR (ROTATIONS, TRANSFORMATIONS, SCALING, VIEWPOINT, MODELLING)

3. ML & AI (ENCODING, MODELS, NEURAL NETWORK, IMAGE RECOGNITION, FNN, RNN, CNN)

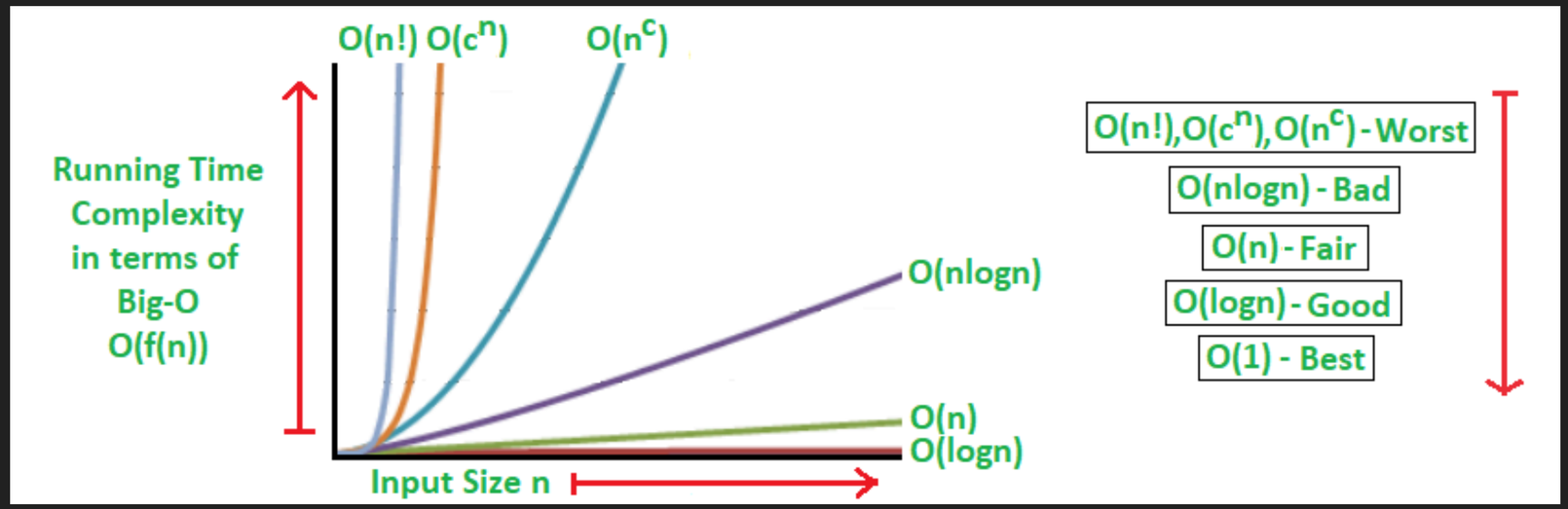
4. ROBOTICS (SENSORY DATA PROCESSING, DYNAMIC MOVEMENT, NAVIGATION)

5. DATA ANALYTICS (PREPROCESSING, OPERATIONS, INTERACTIONS, REAL-TIME DECISION MAKING)

SIGNAL PROCESSING, AND MANY MORE

2. BIG O NOTATION

2. BIG O NOTATION



2. BIG O NOTATION

1. MATHEMATICAL WAY TO DESCRIBE RUNTIME AND SPACE REQUIREMENT OF AN ALGORITHM

2. BIG O NOTATION

1. MATHEMATICAL WAY TO DESCRIBE RUNTIME AND SPACE REQUIREMENT OF AN ALGORITHM

2. DO NOT DEPEND ON SYSTEM.
   WHY?

2. BIG O NOTATION

1. MATHEMATICAL WAY TO DESCRIBE RUNTIME AND SPACE REQUIREMENT OF AN ALGORITHM

2. DO NOT DEPEND ON SYSTEM.
   WHY?
   TIME TAKEN TO EXECUTE SINGLE STEP IS TAKEN AS CONSTANT, O(1)

2. BIG O NOTATION

1. MATHEMATICAL WAY TO DESCRIBE RUNTIME AND SPACE REQUIREMENT OF AN ALGORITHM

2. DO NOT DEPEND ON SYSTEM.
   WHY?
   TIME TAKEN TO EXECUTE SINGLE STEP IS TAKEN AS CONSTANT, O(1)

3. CONSTANT ADDITIONAL AND MULTIPLICATIONS ARE IGNORED AND MOST DOMINATING TERM IS CONSIDERED

2. BIG O NOTATION

1. MATHEMATICAL WAY TO DESCRIBE RUNTIME AND SPACE REQUIREMENT OF AN ALGORITHM

2. DO NOT DEPEND ON SYSTEM.
   WHY?
   TIME TAKEN TO EXECUTE SINGLE STEP IS TAKEN AS CONSTANT, O(1)

3. CONSTANT ADDITIONAL AND MULTIPLICATIONS ARE IGNORED AND MOST DOMINATING TERM IS CONSIDERED

EXAMPLE: O(5N^2 + 3N + 2) =

2. BIG O NOTATION

1. MATHEMATICAL WAY TO DESCRIBE RUNTIME AND SPACE REQUIREMENT OF AN ALGORITHM

2. DO NOT DEPEND ON SYSTEM.
   WHY?
   TIME TAKEN TO EXECUTE SINGLE STEP IS TAKEN AS CONSTANT, O(1)

3. CONSTANT ADDITIONAL AND MULTIPLICATIONS ARE IGNORED AND MOST DOMINATING TERM IS CONSIDERED

EXAMPLE: O(5N^2 + 3N + 2) = O(5N^2) =

2. BIG O NOTATION

1. MATHEMATICAL WAY TO DESCRIBE RUNTIME AND SPACE REQUIREMENT OF AN ALGORITHM

2. DO NOT DEPEND ON SYSTEM.
   WHY?
   TIME TAKEN TO EXECUTE SINGLE STEP IS TAKEN AS CONSTANT, O(1)

3. CONSTANT ADDITIONAL AND MULTIPLICATIONS ARE IGNORED AND MOST DOMINATING TERM IS CONSIDERED

EXAMPLE: O(5N^2 + 3N + 2) = O(5N^2) = O(N^2)

DOUBTS?

TYPES OF MATRIX MULTIPLICATION

TYPES OF MATRIX MULTIPLICATION

C = A X B

Number of Multiplications = p * q * r

TYPES OF MATRIX MULTIPLICATION

Z = A X B X C

C = A X B

Number of Multiplications = p * q * r

TYPES OF MATRIX MULTIPLICATION

Z = A X B X C

Number of Multiplications, (A x B) x C = (p * q * r) + (p * r * s)

Number of Multiplications, A x (B x C) = (q * r * s) + (p * q * s)

TYPES OF MATRIX MULTIPLICATION

Z = A X B X C

Number of Multiplications, (A x B) x C = (p * q * r) + (p * r * s)
Number of Multiplications, A x (B x C) = (q * r * s) + (p * q * s)

E.g., Consider A (9 x 11), B (11 x 13) and C (13 x 15)

Number of Multiplications, (A x B) x C = 3042
Number of Multiplications, A x (B x C) = 3630

TYPES OF MATRIX MULTIPLICATION

Z = A X B X C

Number of Multiplications, (A x B) x C = (p * q * r) + (p * r * s)

Number of Multiplications, A x (B x C) = (q * r * s) + (p * q * s)

E.g., Consider A (9 x 11), B (11 x 13) and C (13 x 15)

Number of Multiplications, (A x B) x C = 3042 ✓

Number of Multiplications, A x (B x C) = 3630

ORDER OF MATRIX MULTIPLICATION

ORDER OF MATRIX MULTIPLICATION

```
// Initialize cost for a single matrix multiplication.
for i from 1 to n do
    m[i][i] ← 0

// l is the chain length (number of matrices in the subchain).
for l from 2 to n do
    for i from 1 to n - l + 1 do
        j ← i + l - 1
        m[i][j] ← ∞
         for k from i to j - 1 do
            // Cost = cost of multiplying matrices A_i..A_k +
            //        cost of multiplying matrices A_(k+1)..A_j +
            //        cost of multiplying the two resulting matrices
            cost ← m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j]
            if cost < m[i][j] then
                m[i][j] ← cost
                s[i][j] ← k
 return (m, s)
```

ORDER OF MATRIX MULTIPLICATION

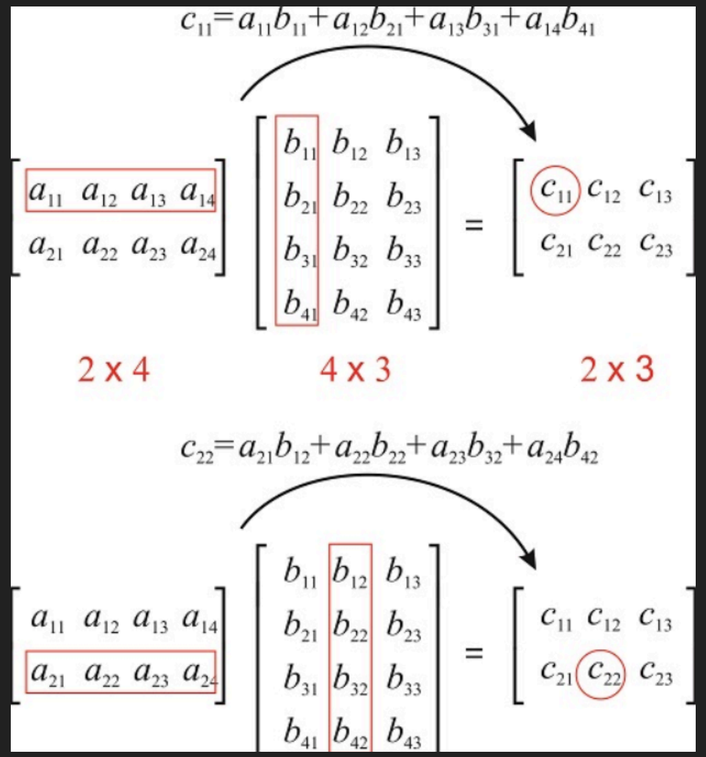
```
// Initialize cost for a single matrix multiplication.
for i from 1 to n do
    m[i][i] ← 0

// l is the chain length (number of matrices in the subchain).
for l from 2 to n do
    for i from 1 to n - l + 1 do
        j ← i + l - 1
        m[i][j] ← ∞
         for k from i to j - 1 do
            // Cost = cost of multiplying matrices A_i..A_k +
            //        cost of multiplying matrices A_(k+1)..A_j +
            //        cost of multiplying the two resulting matrices
            cost ← m[i][k] + m[k+1][j] + p[i-1] * p[k] * p[j]
            if cost < m[i][j] then
                m[i][j] ← cost
                s[i][j] ← k
 return (m, s)
```

NOW, PROBLEM IS OPTIMISING BINARY MULTIPLICATION

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

APPROACH



3. NAIVE APPROACH OF MATRIX MULTIPLICATION

```
n ← number of columns in A       // Also equals number of rows in B
p ← number of columns in B

// Create an m x p result matrix C, initializing every element to 0.
Allocate matrix C of dimensions m x p and set every C[i][j] ← 0

// Loop over each row of A and each column of B.
for i from 0 to m - 1 do:
     for j from 0 to p - 1 do:
         // Compute the (i, j)-th entry of the result.
         for k from 0 to n - 1 do:
             C[i][j] ← C[i][j] + A[i][k] * B[k][j]

 return C
```

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

```
n ← number of columns in A        // Also equals number of rows in B
p ← number of columns in B

// Create an m x p result matrix C, initializing every element to 0.
Allocate matrix C of dimensions m x p and set every C[i][j] ← 0

// Loop over each row of A and each column of B.
for i from 0 to m - 1 do:
     for j from 0 to p - 1 do:
         // Compute the (i, j)-th entry of the result.
         for k from 0 to n - 1 do:
             C[i][j] ← C[i][j] + A[i][k] * B[k][j]

 return C
```

TIME: O(M * N * P) = O(N^3)

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

```
n ← number of columns in A        // Also equals number of rows in B
p ← number of columns in B

// Create an m x p result matrix C, initializing every element to 0.
Allocate matrix C of dimensions m x p and set every C[i][j] ← 0

// Loop over each row of A and each column of B.
for i from 0 to m - 1 do:
     for j from 0 to p - 1 do:
         // Compute the (i, j)-th entry of the result.
         for k from 0 to n - 1 do:
             C[i][j] ← C[i][j] + A[i][k] * B[k][j]

 return C
```

TIME: O(M * N * P) = O(N^3)

SPACE = O(M * P) = O(N^2)

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

HOW IT BEHAVE IN LANGUAGES?

SAY, PYTHON

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

```python
def naive_matrix_multiplication(A, B):
    rows_A, cols_A = len(A), len(A[0])
    rows_B, cols_B = len(B), len(B[0])

    if cols_A != rows_B:
        raise ValueError("Number of columns in A must be equal to number of rows in B")

    # Initialize result matrix with zeros
    result = [[0] * cols_B for _ in range(rows_A)]

    # Perform matrix multiplication
    for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                result[i][j] += A[i][k] * B[k][j]

    return result
```

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

```python
def naive_matrix_multiplication(A, B):
    rows_A, cols_A = len(A), len(A[0])
    rows_B, cols_B = len(B), len(B[0])

    if cols_A != rows_B:
        raise ValueError("Number of columns in A must be equal to number of rows in B")

    # Initialize result matrix with zeros
    result = [[0] * cols_B for _ in range(rows_A)]

    # Perform matrix multiplication
    for i in range(rows_A):
        for j in range(cols_B):
            for k in range(cols_A):
                result[i][j] += A[i][k] * B[k][j]

    return result
```

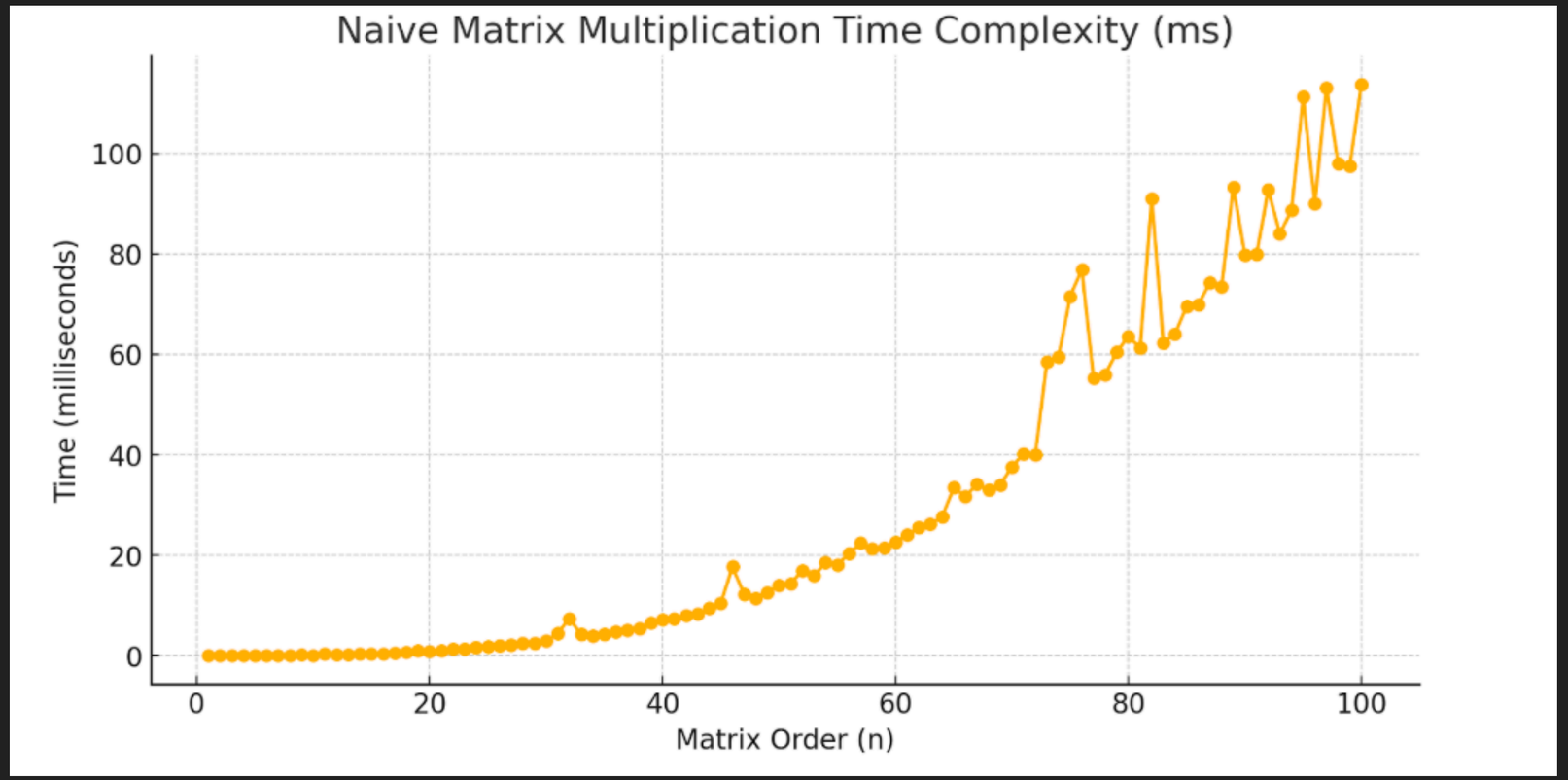
TIME: O(M * N * P) = O(N^3)

SPACE = O(M * P) = O(N^2)

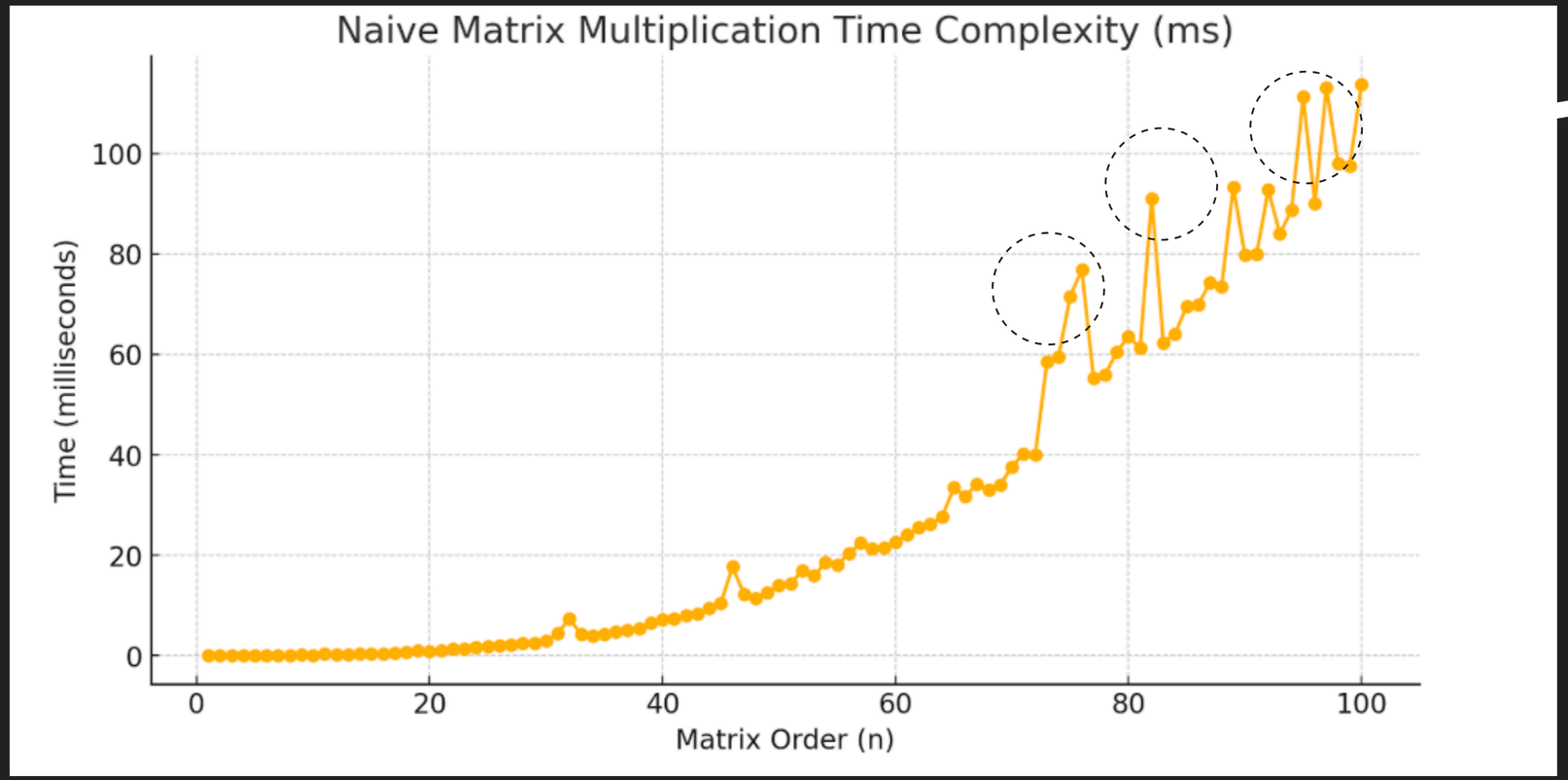
3. NAIVE APPROACH OF MATRIX MULTIPLICATION

WHAT ABOUT TIME REQUIRED IN REAL LIFE?

3. NAIVE APPROACH OF MATRIX MULTIPLICATION



3. NAIVE APPROACH OF MATRIX MULTIPLICATION



DOUBT?

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

WHY SUDDEN SPIKES OF RUNTIME IN BETWEEN?

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

WHY SUDDEN SPIKES OF RUNTIME IN BETWEEN?

1. CPU CACHING EFFECTS: WHEN THE MATRIX SIZE INCREASES, IT MAY SUDDENLY EXCEED DIFFERENT LEVELS OF CPU CACHE (L1, L2, L3), LEADING TO INCREASED MEMORY ACCESS TIME.

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

WHY SUDDEN SPIKES OF RUNTIME IN BETWEEN?

1. CPU CACHING EFFECTS: WHEN THE MATRIX SIZE INCREASES, IT MAY SUDDENLY EXCEED DIFFERENT LEVELS OF CPU CACHE (L1, L2, L3), LEADING TO INCREASED MEMORY ACCESS TIME.

2. MEMORY MANAGEMENT AND PAGING: IF THE MATRIX SIZE GROWS BEYOND WHAT CAN BE COMFORTABLY HELD IN RAM, THE SYSTEM MIGHT START USING VIRTUAL MEMORY, CAUSING OCCASIONAL SLOWDOWNS.

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

WHY SUDDEN SPIKES OF RUNTIME IN BETWEEN?

1. CPU CACHING EFFECTS: WHEN THE MATRIX SIZE INCREASES, IT MAY SUDDENLY EXCEED DIFFERENT LEVELS OF CPU CACHE (L1, L2, L3), LEADING TO INCREASED MEMORY ACCESS TIME.

2. MEMORY MANAGEMENT AND PAGING: IF THE MATRIX SIZE GROWS BEYOND WHAT CAN BE COMFORTABLY HELD IN RAM, THE SYSTEM MIGHT START USING VIRTUAL MEMORY, CAUSING OCCASIONAL SLOWDOWNS.

3. GARBAGE COLLECTION (IF USING PYTHON OR JAVA): IN LANGUAGES WITH AUTOMATIC MEMORY MANAGEMENT, GARBAGE COLLECTION MIGHT RUN AT UNPREDICTABLE TIMES, CAUSING MOMENTARY SPIKES IN EXECUTION TIME.

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

WHAT PERCENT OF DECREASE IN TIME MAY HAPPEN IF WE CHANGE THE LANGUAGE ?

GUESS ?

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

LET'S SWITCH TO C / C++ AS THE PROGRAMMING LANGUAGE

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

```
vector<vector<int>> naiveMatrixMultiplication(const vector<vector<int>>& A, const vecto
    vector<vector<int>> result(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}
```

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

```
vector<vector<int>> naiveMatrixMultiplication(const vector<vector<int>>& A, const vecto
    vector<vector<int>> result(n, vector<int>(n, 0));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return result;
}
```

TIME: O(M * N * P) = O(N^3)

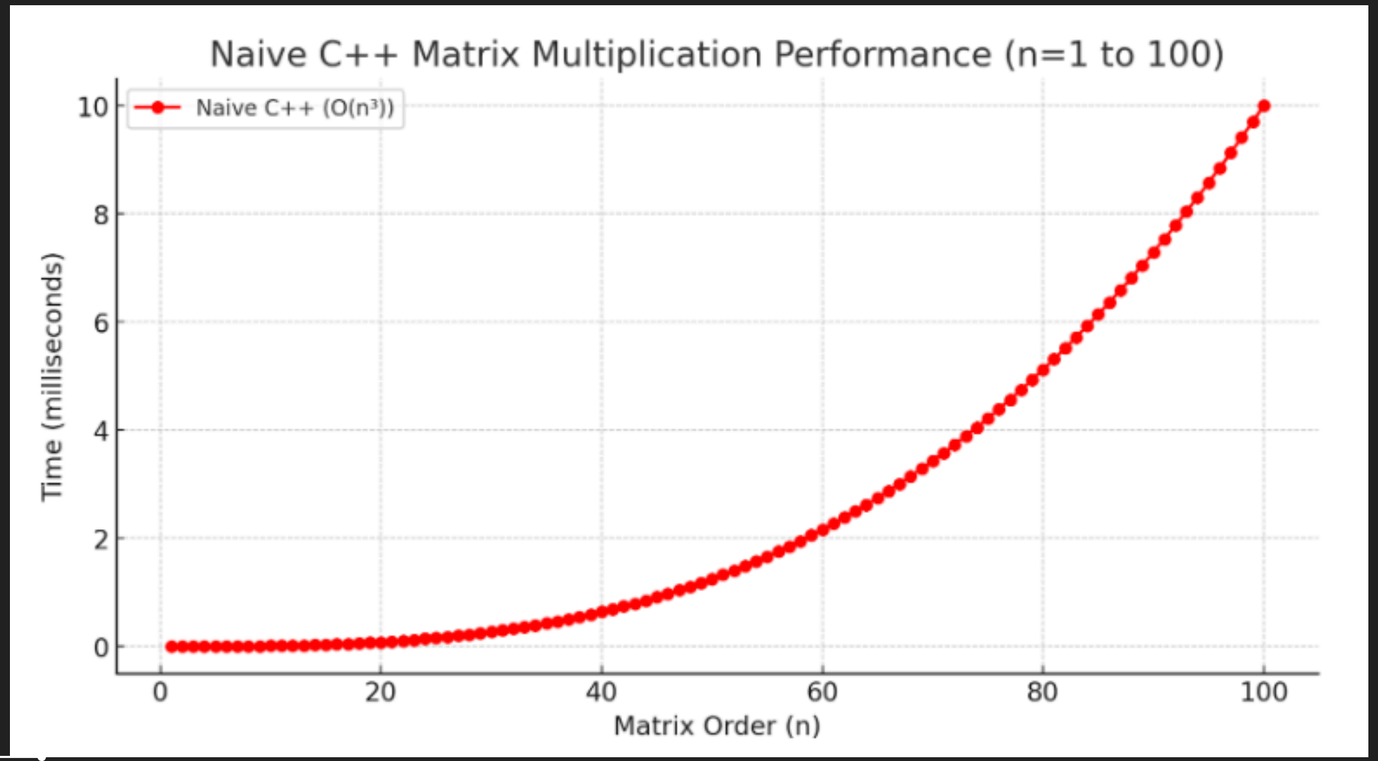
SPACE = O(M * P) = O(N^2)

3. NAIVE APPROACH OF MATRIX MULTIPLICATION

LET'S LOOK INTO THE TIME V/S N GRAPH

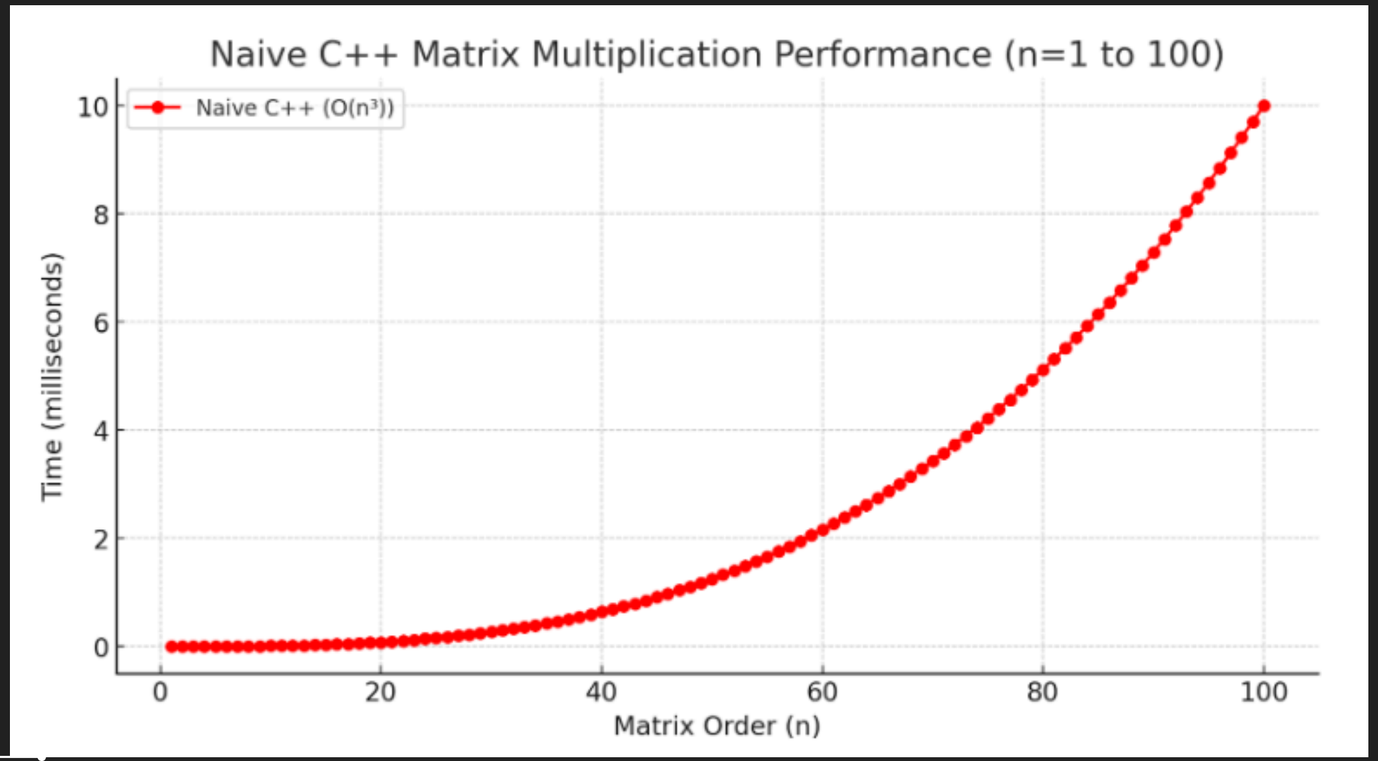
3. NAIVE APPROACH OF MATRIX MULTIPLICATION

LET'S LOOK INTO THE TIME V/S N GRAPH



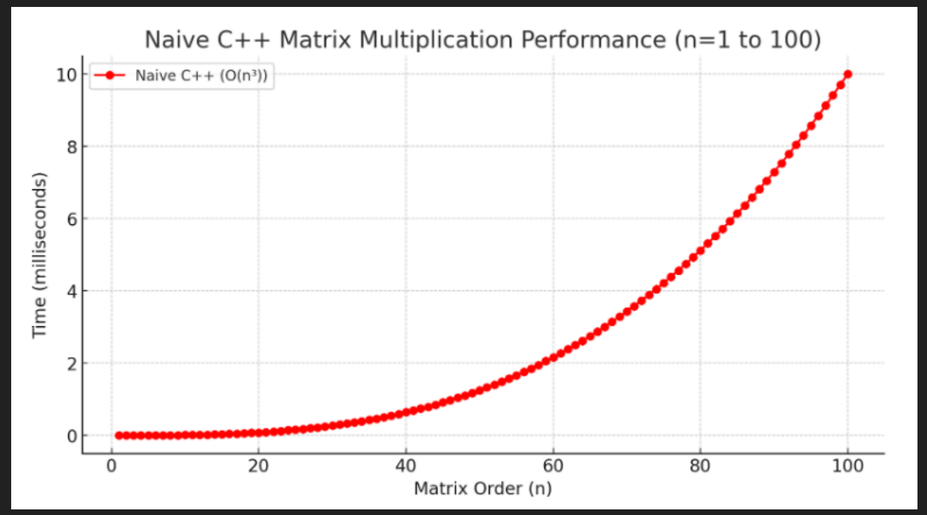
3. NAIVE APPROACH OF MATRIX MULTIPLICATION

LET'S LOOK INTO THE TIME V/S N GRAPH

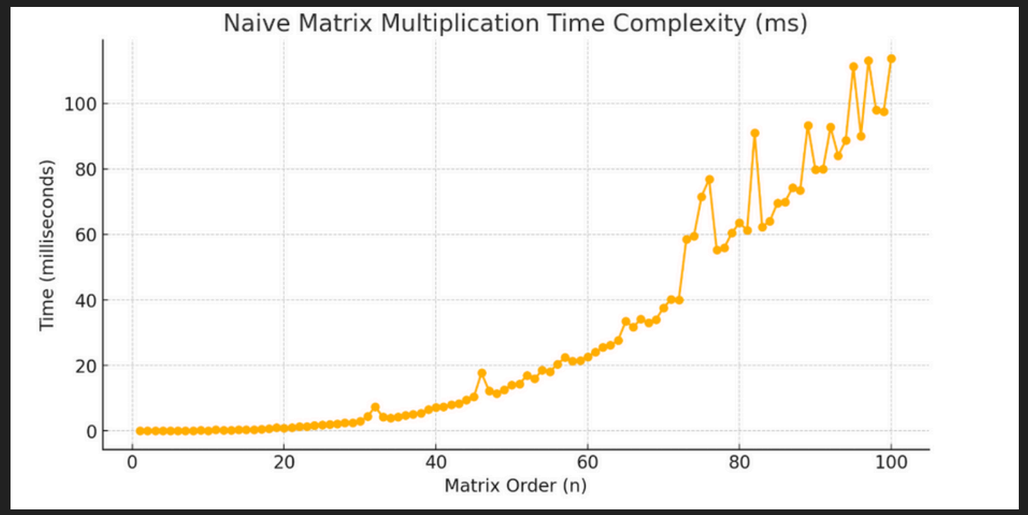


SEE THE DIFFERENCE

3. NAIVE APPROACH OF MATRIX MULTIPLICATION



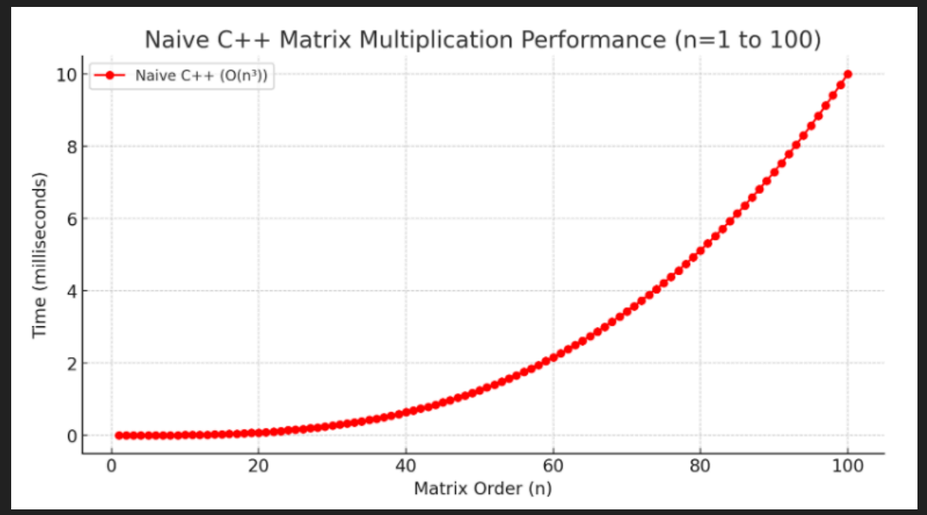
C++

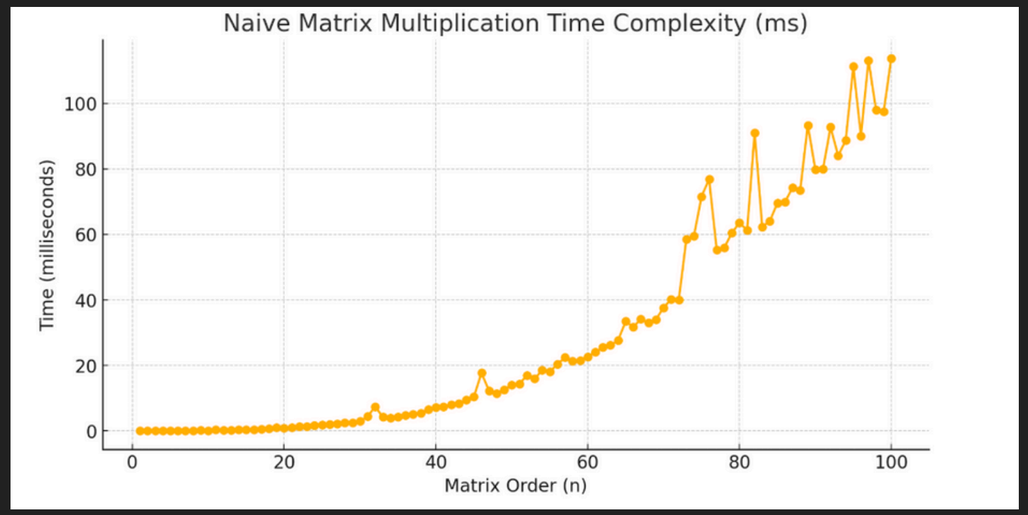


PYTHON

NEARLY 10X FASTER

3. NAIVE APPROACH OF MATRIX MULTIPLICATION





C++

PYTHON

NEARLY 10X FASTER

DOUBT

WHY C / C++ HAS 10 TIMES FASTER SPEED THAN PYTHON?

WHY C / C++ HAS 10 TIMES FASTER SPEED THAN PYTHON?

1. COMPILER V/S INTERPRETER:

   IN C++, CODE IS DIRECTLY CONVERTED TO MACHINE LANGUAGE, LEADING TO OPTIMISED CPU EXECUTION

   DUE TO THIS, C++ ALREADY KNOW WHICH VARIABLES ARE COMING DURING COMPLETER EXECUTION OF THE PROGRAM.

   PYTHON DYNAMICALLY ALLOCATE MEMORY FOR VARIABLES AND THUS, REQUIRE MULTIPLE REALLOC COMMANDS LEADING TO DECREASED SPEED.

WHY C / C++ HAS 10 TIMES FASTER SPEED THAN PYTHON?

1. COMPILER V/S INTERPRETER:
   IN C++, CODE IS DIRECTLY CONVERTED TO MACHINE LANGUAGE, LEADING TO OPTIMISED CPU EXECUTION
   DUE TO THIS, C++ ALREADY KNOW WHICH VARIABLES ARE COMING DURING COMPLETER EXECUTION OF THE PROGRAM.
   PYTHON DYNAMICALLY ALLOCATE MEMORY FOR VARIABLES AND THUS, REQUIRE MULTIPLE REALLOC COMMANDS LEADING TO DECREASED SPEED.

2. LOOP COUNTERS IN C++ IS A INTEGER, WHICH OPERATES FASTER, BUT IN PYTHON, LOOP COUNTER IS A OBJECT WHICH REQUIRE ADDITIONAL CHECKS AND MEMORY ACCESS.

WHY C / C++ HAS 10 TIMES FASTER SPEED THAN PYTHON?

1. COMPILER V/S INTERPRETER:
   IN C++, CODE IS DIRECTLY CONVERTED TO MACHINE LANGUAGE, LEADING TO OPTIMISED CPU EXECUTION
   DUE TO THIS, C++ ALREADY KNOW WHICH VARIABLES ARE COMING DURING COMPLETER EXECUTION OF THE PROGRAM.
   PYTHON DYNAMICALLY ALLOCATE MEMORY FOR VARIABLES AND THUS, REQUIRE MULTIPLE REALLOC COMMANDS LEADING TO DECREASED SPEED.

2. LOOP COUNTERS IN C++ IS A INTEGER, WHICH OPERATES FASTER, BUT IN PYTHON, LOOP COUNTER IS A OBJECT WHICH REQUIRE ADDITIONAL CHECKS AND MEMORY ACCESS.

3. DUE TO STATIC NATURE, FUNCTION CALLS ARE ASSOCIATED IN COMPILATION TIME IN C++, WHICH LEADS TO BETTER MEMORY MANAGEMENT.
   IN PYTHON, DUE TO ITS DYNAMIC NATURE, FUNCTION CALLS ARE EXPENSIVE.

HOW TO INCREASE SPEED OF MATRIX MULTIPLICATION ?

HOW TO INCREASE SPEED OF MATRIX MULTIPLICATION ?

HOW TO INCREASE SPEED OF MATRIX MULTIPLICATION ?

ALGORITHMIC ENHANCEMENTS

HOW TO INCREASE SPEED OF MATRIX MULTIPLICATION ?

ALGORITHMIC ENHANCEMENTS

HOW TO INCREASE SPEED OF MATRIX MULTIPLICATION ?

ALGORITHMIC ENHANCEMENTS

COMPILER OPTIMIZATIONS AND HARDWARE ACCELERATION

4. ALGORITHMIC ENHANCEMENTS

4. ALGORITHMIC ENHANCEMENTS

STRASSEN'S ALGORITHM

4. ALGORITHMIC ENHANCEMENTS

STRASSEN'S ALGORITHM

THERE ARE SOME MORE ALGORITHMS WHICH ARE NOTHING BUT THE VARIANTS OF STRASSEN ALGORITHM WITH SLIGHT TWEEKS IN VARIABLE DECLARATIONS AND LESS RUNNING STATEMENTS, BUT OVERALL NO CONSIDERABLE CHANGE IS BROUGHT BY THEM.

STRASSEN ALGORITHM

STRASSEN ALGORITHM

$$p1 = a(f - h) \qquad p2 = (a + b)h$$
$$p3 = (c + d)e \qquad p4 = d(g - e)$$
$$p5 = (a + d)(e + h) \qquad p6 = (b - d)(g + h)$$
$$p7 = (a - c)(e + f)$$

The A x B can be calculated using above seven multiplications.
Following are values of four sub-matrices of result C

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

A, B and C are square metrices of size N x N
a, b, c and d are submatrices of A, of size N/2 x N/2
e, f, g and h are submatrices of B, of size N/2 x N/2
p1, p2, p3, p4, p5, p6 and p7 are submatrices of size N/2 x N/2

STRASSEN ALGORITHM

```
Function StrassenMultiply(A, B, n):
    If n == 1:  # Base case: 1x1 matrix
        Return A[0][0] * B[0][0]

    # Divide A and B into submatrices
    A11, A12, A21, A22 = SplitMatrix(A)
    B11, B12, B21, B22 = SplitMatrix(B)

    # Compute 7 matrix products (M1 to M7)
    M1 = StrassenMultiply(A11 + A22, B11 + B22)
    M2 = StrassenMultiply(A21 + A22, B11)
    M3 = StrassenMultiply(A11, B12 - B22)
    M4 = StrassenMultiply(A22, B21 - B11)
    M5 = StrassenMultiply(A11 + A12, B22)
    M6 = StrassenMultiply(A21 - A11, B11 + B12)
    M7 = StrassenMultiply(A12 - A22, B21 + B22)
```

STRASSEN ALGORITHM

```
    # Compute the result submatrices
    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    # Combine submatrices to form the final matrix C
    C = MergeMatrices(C11, C12, C21, C22)

    Return C

Function SplitMatrix(Matrix):
    # Divide matrix into 4 equal parts and return them
    Return (Top-Left, Top-Right, Bottom-Left, Bottom-Right)

Function MergeMatrices(C11, C12, C21, C22):
    # Combine the four submatrices into one
    Return CombinedMatrix
```

STRASSEN ALGORITHM

```
    # Compute the result submatrices
    C11 = M1 + M4 - M5 + M7
    C12 = M3 + M5
    C21 = M2 + M4
    C22 = M1 - M2 + M3 + M6

    # Combine submatrices to form the final matrix C
    C = MergeMatrices(C11, C12, C21, C22)

    Return C

Function SplitMatrix(Matrix):
    # Divide matrix into 4 equal parts and return them
    Return (Top-Left, Top-Right, Bottom-Left, Bottom-Right)

Function MergeMatrices(C11, C12, C21, C22):
    # Combine the four submatrices into one
    Return CombinedMatrix
```

TIME: O(N^LOG(7)) = O(N^2.81)

SPACE = O(N^2)

CAN WE ACHIEVE SOMETHING MORE BETTER ALGORITHMICALLY ?

CAN WE ACHIEVE SOMETHING MORE BETTER ALGORITHMICALLY ?

YES, WE CAN ACHIEVE O(N^2.376) , BUT ……

CAN WE ACHIEVE SOMETHING MORE BETTER ALGORITHMICALLY ?

YES, WE CAN ACHIEVE O(N^2.376) , BUT ……

ONLY THEORETICALLY

CAN WE ACHIEVE SOMETHING MORE BETTER ALGORITHMICALLY ?

YES, WE CAN ACHIEVE O(N^2.376) , BUT ……

ONLY THEORETICALLY

COPPERSMITH - WINOGRAD VARIANT OF STRASSEN

COPPERSMITH - WINOGRAD VARIANT OF STRASSEN

COPPERSMITH - WINOGRAD VARIANT OF STRASSEN

1. USES TENSOR-BASED ALGEBRA TO OPTIMIZE MATRIX MULTIPLICATION.

COPPERSMITH - WINOGRAD VARIANT OF STRASSEN

1. USES TENSOR-BASED ALGEBRA TO OPTIMIZE MATRIX MULTIPLICATION.
2. REDUCES THE NUMBER OF SCALAR MULTIPLICATIONS NEEDED.

COPPERSMITH - WINOGRAD VARIANT OF STRASSEN

1. USES TENSOR-BASED ALGEBRA TO OPTIMIZE MATRIX MULTIPLICATION.
2. REDUCES THE NUMBER OF SCALAR MULTIPLICATIONS NEEDED.
3. STRASSEN'S METHOD REDUCES MULTIPLICATIONS FROM 8 TO 7 PER RECURSIVE STEP; COPPERSMITH-WINOGRAD IMPROVES THIS FURTHER.

COPPERSMITH - WINOGRAD VARIANT OF STRASSEN

1. USES TENSOR-BASED ALGEBRA TO OPTIMIZE MATRIX MULTIPLICATION.
2. REDUCES THE NUMBER OF SCALAR MULTIPLICATIONS NEEDED.
3. STRASSEN'S METHOD REDUCES MULTIPLICATIONS FROM 8 TO 7 PER RECURSIVE STEP; COPPERSMITH-WINOGRAD IMPROVES THIS FURTHER.

WHY NOT BETTER IN REAL LIFE APPLICATIONS ?

DOUBT

COPPERSMITH - WINOGRAD VARIANT OF STRASSEN

WHY NOT BETTER IN REAL LIFE APPLICATIONS ?

COPPERSMITH - WINOGRAD VARIANT OF STRASSEN

✅ FASTER IN THEORY, ESPECIALLY FOR HUGE MATRICES

COPPERSMITH - WINOGRAD VARIANT OF STRASSEN

✅ FASTER IN THEORY, ESPECIALLY FOR HUGE MATRICES

❌ HIGH CONSTANT FACTORS MAKE IT INEFFICIENT FOR SMALL OR MODERATE SIZES

COPPERSMITH - WINOGRAD VARIANT OF STRASSEN

✅ FASTER IN THEORY, ESPECIALLY FOR HUGE MATRICES

❌ HIGH CONSTANT FACTORS MAKE IT INEFFICIENT FOR SMALL OR MODERATE SIZES

❌ NOT WIDELY USED IN PRACTICE DUE TO COMPLEXITY

COPPERSMITH - WINOGRAD VARIANT OF STRASSEN

✅ FASTER IN THEORY, ESPECIALLY FOR HUGE MATRICES
❌ HIGH CONSTANT FACTORS MAKE IT INEFFICIENT FOR SMALL OR MODERATE SIZES
❌ NOT WIDELY USED IN PRACTICE DUE TO COMPLEXITY

FOR MOST REAL-WORLD APPLICATIONS, STRASSEN'S ALGORITHM OR HIGHLY OPTIMIZED CLASSICAL MATRIX MULTIPLICATION (BLAS, GPUS) IS PREFERRED OVER COPPERSMITH-WINOGRAD.

5. HARDWARE DRIVEN AND COMPILER OPTIMIZATIONS

5. HARDWARE DRIVEN AND COMPILER OPTIMIZATIONS

5. HARDWARE DRIVEN AND COMPILER OPTIMIZATIONS

1. CACHE BLOCKING

5. HARDWARE DRIVEN AND COMPILER OPTIMIZATIONS

1. CACHE BLOCKING
2. LOOP UNROLLING

5. HARDWARE DRIVEN AND COMPILER OPTIMIZATIONS

1. CACHE BLOCKING
2. LOOP UNROLLING
3. AGGRESSIVE COMPILER OPTIMIZATIONS

5. HARDWARE DRIVEN AND COMPILER OPTIMIZATIONS

1. CACHE BLOCKING
2. LOOP UNROLLING
3. AGGRESSIVE COMPILER OPTIMIZATIONS
4. SIMD (VECTORIZATION)

5. HARDWARE DRIVEN AND COMPILER OPTIMIZATIONS

1. CACHE BLOCKING
2. LOOP UNROLLING
3. AGGRESSIVE COMPILER OPTIMIZATIONS
4. SIMD (VECTORIZATION)
5. MULTI-THREADING AND PARALLELISM

CACHE BLOCKING

CACHE BLOCKING

MODERN CPUS HAVE HIERARCHICAL MEMORY :

1. REGISTERS (FASTEST, BUT SMALLEST)
2. L1 CACHE (SMALL, BUT VERY FAST)
3. L2 CACHE (LARGER BUT SLOWER THAN L1)
4. L3 CACHE (EVEN LARGER, SHARED ACROSS CORES)
5. RAM (MAIN MEMORY) (MUCH SLOWER)

CACHE BLOCKING

MODERN CPUS HAVE HIERARCHICAL MEMORY :

1. REGISTERS (FASTEST, BUT SMALLEST)
2. L1 CACHE (SMALL, BUT VERY FAST)
3. L2 CACHE (LARGER BUT SLOWER THAN L1)
4. L3 CACHE (EVEN LARGER, SHARED ACROSS CORES)
5. RAM (MAIN MEMORY) (MUCH SLOWER)

PROBLEM WITH STANDARD MATRIX MULTIPLICATION :

❌ A LARGE N×N MATRIX DOES NOT FIT IN L1/L2 CACHE.

❌ ACCESSING ELEMENTS IN A NON-SEQUENTIAL ORDER CAUSES CACHE MISSES.

❌ THE CPU KEEPS FETCHING DATA FROM RAM, SLOWING EXECUTION.

CACHE BLOCKING

MODERN CPUS HAVE HIERARCHICAL MEMORY :

1. REGISTERS (FASTEST, BUT SMALLEST)
2. L1 CACHE (SMALL, BUT VERY FAST)
3. L2 CACHE (LARGER BUT SLOWER THAN L1)
4. L3 CACHE (EVEN LARGER, SHARED ACROSS CORES)
5. RAM (MAIN MEMORY) (MUCH SLOWER)

PROBLEM WITH STANDARD MATRIX MULTIPLICATION :

❌ A LARGE N×N MATRIX DOES NOT FIT IN L1/L2 CACHE.
❌ ACCESSING ELEMENTS IN A NON-SEQUENTIAL ORDER CAUSES CACHE MISSES.
❌ THE CPU KEEPS FETCHING DATA FROM RAM, SLOWING EXECUTION.

SOLUTION → CACHE BLOCKING :

✅ DIVIDE THE MATRIX INTO SMALLER "BLOCKS" (TILES) OF SIZE B×B.
✅ WORK ON EACH BLOCK ENTIRELY IN CACHE BEFORE MOVING TO THE NEXT.
✅ REDUCES CACHE MISSES AND IMPROVES MEMORY ACCESS EFFICIENCY.

LOOP UNROLLING

LOOP UNROLLING

WE REDUCE THE NUMBER OF LOOP ITERATIONS BY EXECUTING MULTIPLE OPERATIONS IN A SINGLE ITERATION. THIS REDUCES:

1. LOOP OVERHEAD (LESS BRANCHING AND INDEX UPDATES).
2. BRANCH MISPREDICTIONS (FEWER JUMPS).
3. INSTRUCTION DEPENDENCY DELAYS (MORE INSTRUCTIONS EXECUTED IN PARALLEL).

LOOP UNROLLING

WE REDUCE THE NUMBER OF LOOP ITERATIONS BY EXECUTING MULTIPLE OPERATIONS IN A SINGLE ITERATION. THIS REDUCES:

1. LOOP OVERHEAD (LESS BRANCHING AND INDEX UPDATES).
2. BRANCH MISPREDICTIONS (FEWER JUMPS).
3. INSTRUCTION DEPENDENCY DELAYS (MORE INSTRUCTIONS EXECUTED IN PARALLEL).

WHY USEFUL ?

- MINIMIZES LOOP CONTROL OVERHEAD:
- INSTEAD OF CHECKING LOOP CONDITIONS IN EVERY ITERATION, WE CHECK LESS FREQUENTLY.
- INCREASES INSTRUCTION-LEVEL PARALLELISM (ILP):
- MORE COMPUTATIONS HAPPEN IN A SINGLE ITERATION, KEEPING CPU EXECUTION UNITS BUSY.
- IMPROVES CACHE EFFICIENCY.
- MORE DATA IS PROCESSED BEFORE RELOADING FROM MEMORY.

SIMD (VECTORIZATION)

SIMD (VECTORIZATION)

SIMD (SINGLE INSTRUCTION, MULTIPLE DATA) IS A PARALLEL PROCESSING TECHNIQUE WHERE A SINGLE CPU INSTRUCTION OPERATES ON MULTIPLE DATA ELEMENTS SIMULTANEOUSLY.

- INSTEAD OF PROCESSING ONE VALUE AT A TIME, SIMD PROCESSES MULTIPLE VALUES IN ONE INSTRUCTION.
- THIS SPEEDS UP TASKS LIKE MATRIX MULTIPLICATION, IMAGE PROCESSING, AND DEEP LEARNING.

SIMD (VECTORIZATION)

SIMD (SINGLE INSTRUCTION, MULTIPLE DATA) IS A PARALLEL PROCESSING TECHNIQUE WHERE A SINGLE CPU INSTRUCTION OPERATES ON MULTIPLE DATA ELEMENTS SIMULTANEOUSLY.

- INSTEAD OF PROCESSING ONE VALUE AT A TIME, SIMD PROCESSES MULTIPLE VALUES IN ONE INSTRUCTION.
- THIS SPEEDS UP TASKS LIKE MATRIX MULTIPLICATION, IMAGE PROCESSING, AND DEEP LEARNING.

Instruction Set	Register Size	Data Processed per Instruction
SSE (Streaming SIMD Extensions)	128-bit	4 floats / 2 doubles
AVX (Advanced Vector Extensions)	256-bit	8 floats / 4 doubles
AVX512	512-bit	16 floats / 8 doubles
NEON (ARM SIMD)	128-bit	4 floats / 2 doubles

AGGRESSIVE COMPILER OPTIMIZATIONS

AGGRESSIVE COMPILER OPTIMIZATIONS

1. FUNCTION INLINING (-FINLINE-FUNCTIONS) :

   INLINES FUNCTIONS TO REDUCE FUNCTION CALL AND OVERHEAD

AGGRESSIVE COMPILER OPTIMIZATIONS

1. FUNCTION INLINING (-FINLINE-FUNCTIONS) :
   INLINES FUNCTIONS TO REDUCE FUNCTION CALL AND OVERHEAD

2. LOOP OPTIMIZATIONS :
   A. LOOP UNROLLING (-FUNROLL-LOOPS)
   B. LOOP FUSION (-FLOOP-FUSION) – MERGES MULTIPLE LOOPS INTO ONE
   C. LOOP INTERCHANGE (-FLOOP-INTERCHANGE) – REORDERS NESTED LOOPS FOR CACHE EFFICIENCY
   D. LOOP VECTORIZATION (-FTREE-VECTORIZE) – USES SIMD TO PROCESS MULTIPLE DATA POINTS

AGGRESSIVE COMPILER OPTIMIZATIONS

1. FUNCTION INLINING (-FINLINE-FUNCTIONS) :

INLINES FUNCTIONS TO REDUCE FUNCTION CALL AND OVERHEAD

2. LOOP OPTIMIZATIONS :

A. LOOP UNROLLING (-FUNROLL-LOOPS)

B. LOOP FUSION (-FLOOP-FUSION) – MERGES MULTIPLE LOOPS INTO ONE

C. LOOP INTERCHANGE (-FLOOP-INTERCHANGE) – REORDERS NESTED LOOPS FOR CACHE EFFICIENCY

D. LOOP VECTORIZATION (-FTREE-VECTORIZE) – USES SIMD TO PROCESS MULTIPLE DATA POINTS

3. VECTORIZATION (SIMD) (-FTREE-VECTORIZE) :

- USES AVX, SSE, OR NEON FOR PARALLEL COMPUTATION.
- PROCESSES MULTIPLE VALUES PER CPU INSTRUCTION.

AGGRESSIVE COMPILER OPTIMIZATIONS

4. FAST FLOATING-POINT MATH (-FFAST-MATH, -FNO-MATH-ERRNO) :

- IGNORES STRICT IEEE FLOATING-POINT RULES TO OPTIMIZE SPEED.
- REMOVES UNNECESSARY CHECKS (E.G., DIVISION BY ZERO).

AGGRESSIVE COMPILER OPTIMIZATIONS

4. FAST FLOATING-POINT MATH (-FFAST-MATH, -FNO-MATH-ERRNO) :

- IGNORES STRICT IEEE FLOATING-POINT RULES TO OPTIMIZE SPEED.
- REMOVES UNNECESSARY CHECKS (E.G., DIVISION BY ZERO).

5. LINK TIME OPTIMIZATION (LTO) (-FLTO) :

- OPTIMIZES ACROSS MULTIPLE SOURCE FILES AT LINK-TIME.
- ALLOWS BETTER FUNCTION INLINING AND CONSTANT PROPAGATION.

AGGRESSIVE COMPILER OPTIMIZATIONS

4. FAST FLOATING-POINT MATH (-FFAST-MATH, -FNO-MATH-ERRNO) :

- IGNORES STRICT IEEE FLOATING-POINT RULES TO OPTIMIZE SPEED.
- REMOVES UNNECESSARY CHECKS (E.G., DIVISION BY ZERO).

5. LINK TIME OPTIMIZATION (LTO) (-FLTO) :

- OPTIMIZES ACROSS MULTIPLE SOURCE FILES AT LINK-TIME.
- ALLOWS BETTER FUNCTION INLINING AND CONSTANT PROPAGATION.

6. DATA PREFETCHING (-FPREFETCH-LOOP-ARRAYS) :

- TELLS THE CPU TO PRELOAD DATA INTO CACHE BEFORE IT IS NEEDED.
- REDUCES MEMORY LATENCY IN LARGE ARRAYS AND MATRIX OPERATIONS.

MULTI-THREADING

MULTI-THREADING

FOR FAST AND EFFICIENT MATRIX MULTIPLICATION, WE CAN OPTIMIZE USING:

MULTI-THREADING

FOR FAST AND EFFICIENT MATRIX MULTIPLICATION, WE CAN OPTIMIZE USING:

1 MULTI-THREADING (CPU PARALLELISM USING OPENMP)

MULTI-THREADING

FOR FAST AND EFFICIENT MATRIX MULTIPLICATION, WE CAN OPTIMIZE USING:

1. MULTI-THREADING (CPU PARALLELISM USING OPENMP)
2. SIMD (VECTORIZATION USING AVX/SSE)

MULTI-THREADING

FOR FAST AND EFFICIENT MATRIX MULTIPLICATION, WE CAN OPTIMIZE USING:

1. MULTI-THREADING (CPU PARALLELISM USING OPENMP)
2. SIMD (VECTORIZATION USING AVX/SSE)
3. GPU ACCELERATION (CUDA FOR NVIDIA GPUS)

MULTI-THREADING

FOR FAST AND EFFICIENT MATRIX MULTIPLICATION, WE CAN OPTIMIZE USING:

1. MULTI-THREADING (CPU PARALLELISM USING OPENMP)
2. SIMD (VECTORIZATION USING AVX/SSE) ✅
3. GPU ACCELERATION (CUDA FOR NVIDIA GPUS)

MULTI-THREADING (CPU PARALLELISM USING OPENMP)

MULTI-THREADING (CPU PARALLELISM USING OPENMP)

MULTI-THREADING ALLOWS MULTIPLE CPU CORES TO COMPUTE DIFFERENT PARTS OF THE MATRIX IN PARALLEL.

MULTI-THREADING (CPU PARALLELISM USING OPENMP)

MULTI-THREADING ALLOWS MULTIPLE CPU CORES TO COMPUTE DIFFERENT PARTS OF THE MATRIX IN PARALLEL.

```
void matmul_omp(float A[N][N], float B[N][N], float C[N][N]) {
    #pragma omp parallel for collapse(2) // Multi-threading
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

MULTI-THREADING (CPU PARALLELISM USING OPENMP)

MULTI-THREADING ALLOWS MULTIPLE CPU CORES TO COMPUTE DIFFERENT PARTS OF THE MATRIX IN PARALLEL.

```
void matmul_omp(float A[N][N], float B[N][N], float C[N][N]) {
    #pragma omp parallel for collapse(2) // Multi-threading
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            C[i][j] = 0;
            for (int k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

✅ CPU CORES WORK IN PARALLEL
(E.G., 8 CORES = 8X SPEEDUP)

✅ NO CHANGES TO LOGIC, JUST
ADDED #PRAGMA OMP PARALLEL FOR

COMPILE WITH:
g++ -O3 -fopenmp matmul.cpp -o matmul

HOW TO INCORPORATE ALL THESE IN ALGORITHM ?

HOW TO INCORPORATE ALL THESE IN ALGORITHM ?

BY USING LIBRARIES :

FOR PYTHON, WE CAN USE THE NUMPY LIBRARY
FOR C++, WE WILL USE THE EIGEN LIBRARY

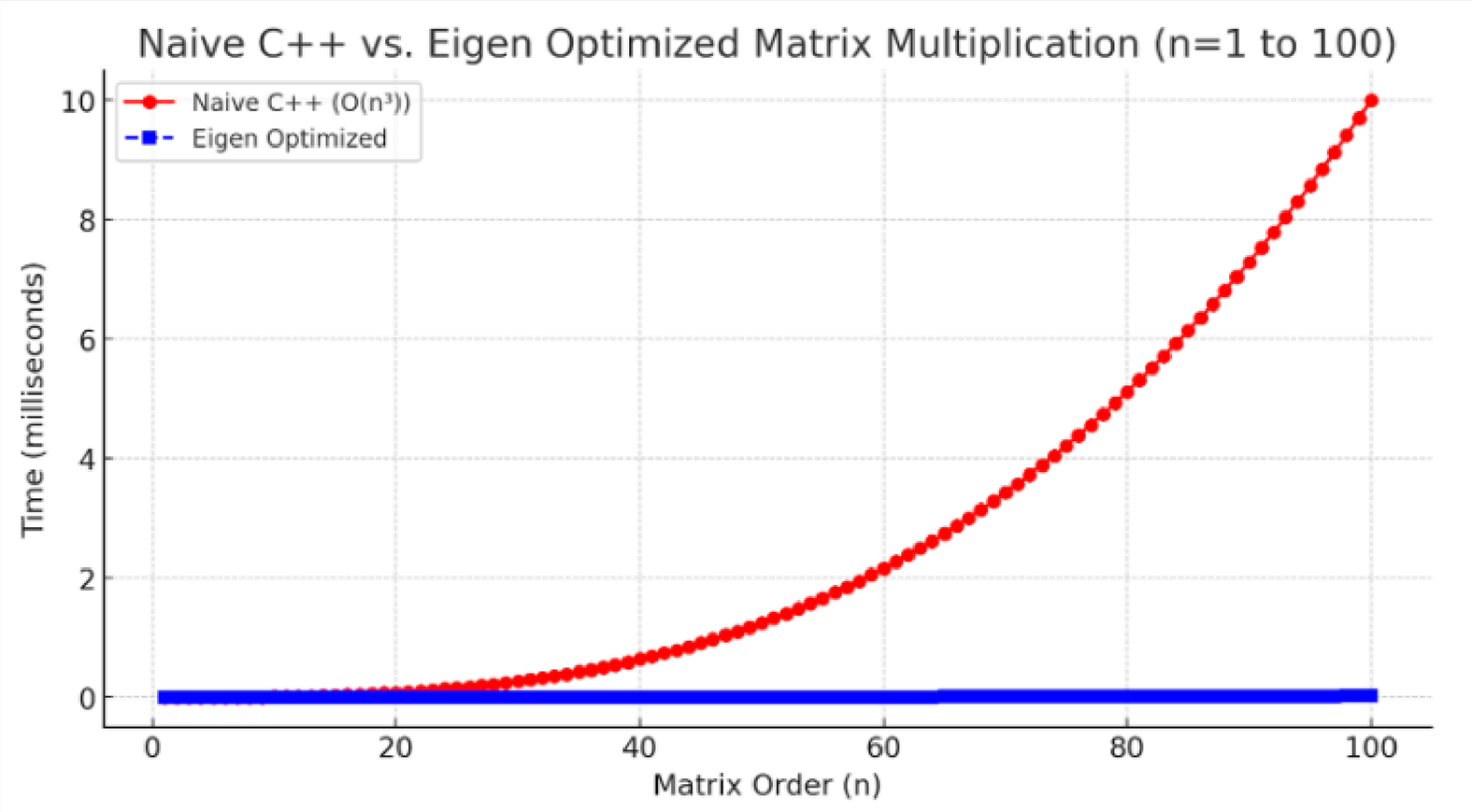
THESE LIBRARIES IMPLEMENT ALL THESE OPTIMISATIONS ON THEIR OWN.

USING EIGEN IN C++

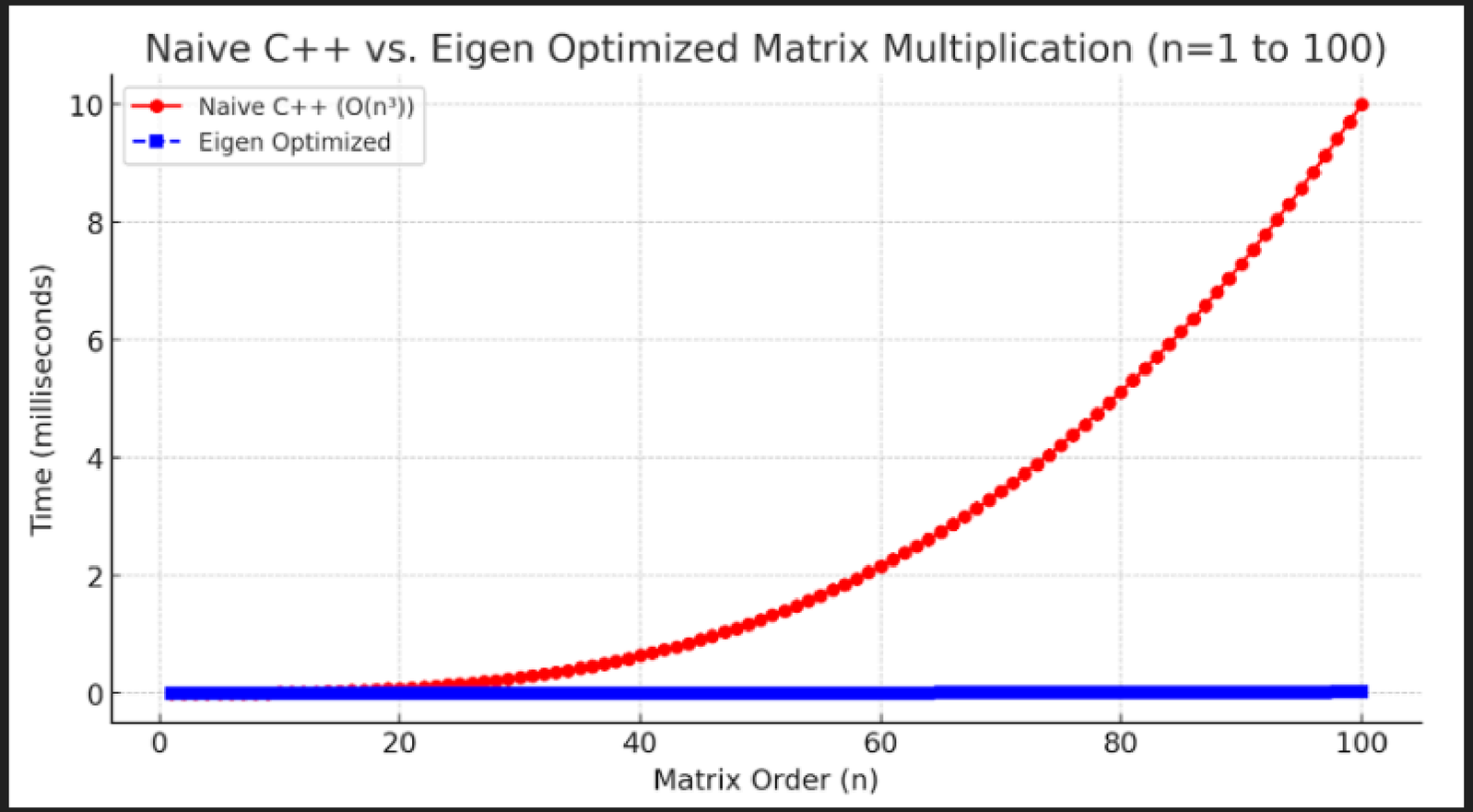
USING EIGEN IN C++

```
// Measure time for Eigen's optimized multiplication
auto start = high_resolution_clock::now();
MatrixXd result = A * B;
auto end = high_resolution_clock::now();
```

USING EIGEN IN C++



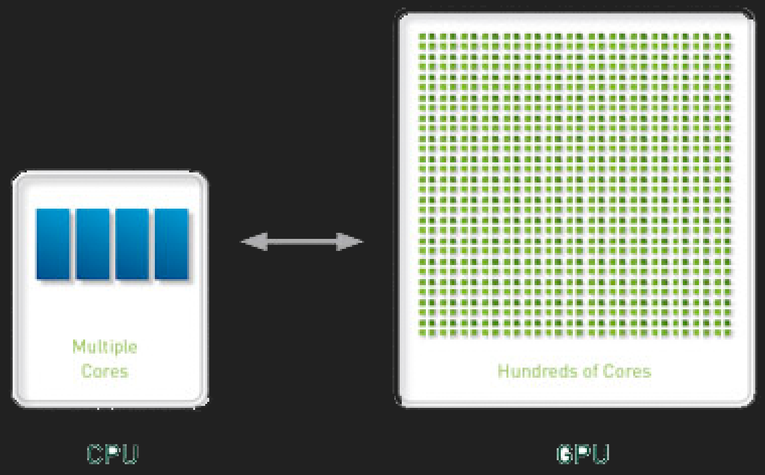
USING EIGEN IN C++



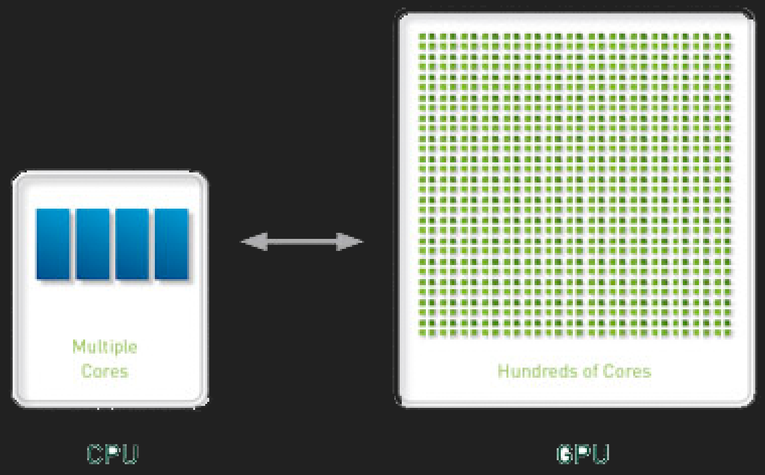
SEE THE DIFFERENCE

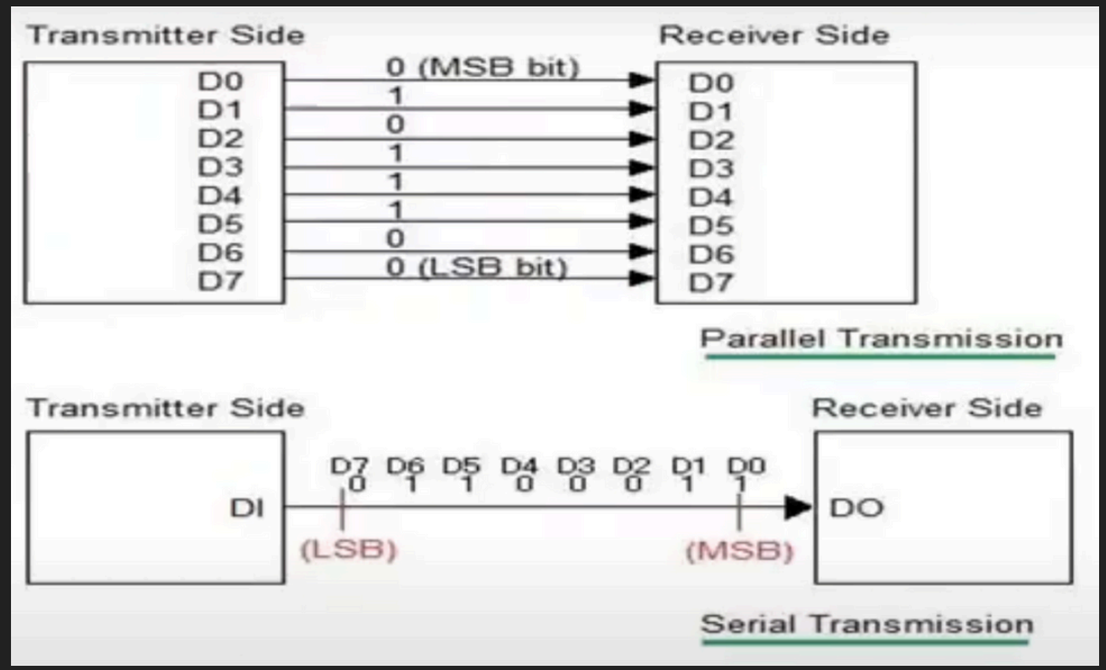
7. GPU ACCELERATION

7. GPU ACCELERATION



7. GPU ACCELERATION





7. GPU ACCELERATION

HOW AND WHERE TO WRITE CODE THAT UTILISE THIS POTENTIAL OF GPU ?

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

ALLOWS DEVELOPERS TO LEVERAGE GPU ACCELERATION FOR GENERAL-PURPOSE COMPUTING. IT PROVIDES AN EXTENSION OF C/C++ WITH SPECIALIZED APIS AND LIBRARIES TO PERFORM HIGH-PERFORMANCE PARALLEL COMPUTATIONS.

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

ALLOWS DEVELOPERS TO LEVERAGE GPU ACCELERATION FOR GENERAL-PURPOSE COMPUTING. IT PROVIDES AN EXTENSION OF C/C++ WITH SPECIALIZED APIS AND LIBRARIES TO PERFORM HIGH-PERFORMANCE PARALLEL COMPUTATIONS.

HOW TO PERFORM ?

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

ALLOWS DEVELOPERS TO LEVERAGE GPU ACCELERATION FOR GENERAL-PURPOSE COMPUTING. IT PROVIDES AN EXTENSION OF C/C++ WITH SPECIALIZED APIS AND LIBRARIES TO PERFORM HIGH-PERFORMANCE PARALLEL COMPUTATIONS.

HOW TO PERFORM ?

1. ALLOCATE MEMORY ON CPU AND GPU

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

ALLOWS DEVELOPERS TO LEVERAGE GPU ACCELERATION FOR GENERAL-PURPOSE COMPUTING. IT PROVIDES AN EXTENSION OF C/C++ WITH SPECIALIZED APIS AND LIBRARIES TO PERFORM HIGH-PERFORMANCE PARALLEL COMPUTATIONS.

HOW TO PERFORM ?

1. ALLOCATE MEMORY ON CPU AND GPU
2. HOST CPU –> GPU DATA TRANSFER

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

ALLOWS DEVELOPERS TO LEVERAGE GPU ACCELERATION FOR GENERAL-PURPOSE COMPUTING. IT PROVIDES AN EXTENSION OF C/C++ WITH SPECIALIZED APIS AND LIBRARIES TO PERFORM HIGH-PERFORMANCE PARALLEL COMPUTATIONS.

HOW TO PERFORM ?

1. ALLOCATE MEMORY ON CPU AND GPU
2. HOST CPU –> GPU DATA TRANSFER
3. DEFINE CUDA KERNEL

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

ALLOWS DEVELOPERS TO LEVERAGE GPU ACCELERATION FOR GENERAL-PURPOSE COMPUTING. IT PROVIDES AN EXTENSION OF C/C++ WITH SPECIALIZED APIS AND LIBRARIES TO PERFORM HIGH-PERFORMANCE PARALLEL COMPUTATIONS.

HOW TO PERFORM ?

1. ALLOCATE MEMORY ON CPU AND GPU
2. HOST CPU –> GPU DATA TRANSFER
3. DEFINE CUDA KERNEL
4. CONFIGURE THREADS AND BLOCKS

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

ALLOWS DEVELOPERS TO LEVERAGE GPU ACCELERATION FOR GENERAL-PURPOSE COMPUTING. IT PROVIDES AN EXTENSION OF C/C++ WITH SPECIALIZED APIS AND LIBRARIES TO PERFORM HIGH-PERFORMANCE PARALLEL COMPUTATIONS.

HOW TO PERFORM ?

1. ALLOCATE MEMORY ON CPU AND GPU
2. HOST CPU –> GPU DATA TRANSFER
3. DEFINE CUDA KERNEL
4. CONFIGURE THREADS AND BLOCKS
5. PARALLEL EXECUTION WITH THOUSANDS OF THREADS

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

ALLOWS DEVELOPERS TO LEVERAGE GPU ACCELERATION FOR GENERAL-PURPOSE COMPUTING. IT PROVIDES AN EXTENSION OF C/C++ WITH SPECIALIZED APIS AND LIBRARIES TO PERFORM HIGH-PERFORMANCE PARALLEL COMPUTATIONS.

HOW TO PERFORM ?

1. ALLOCATE MEMORY ON CPU AND GPU
2. HOST CPU –> GPU DATA TRANSFER
3. DEFINE CUDA KERNEL
4. CONFIGURE THREADS AND BLOCKS
5. PARALLEL EXECUTION WITH THOUSANDS OF THREADS
6. RESULTS TRANSFER BACK TO CPU

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

CPU V/S GPU COMPARISON :

Method	Speedup	Best For
Basic CPU (No Optimization)	1x	Small matrices
OpenMP Multi-threading	4-16x	Medium matrices
AVX + OpenMP (SIMD + Multi-threading)	16-32x	Large matrices
CUDA GPU (Parallel Execution)	50-100x 🚀	Massive matrices (N > 1024)

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

CPU V/S GPU COMPARISON :

Method	Speedup	Best For
Basic CPU (No Optimization)	1x	Small matrices
OpenMP Multi-threading	4-16x	Medium matrices
AVX + OpenMP (SIMD + Multi-threading)	16-32x	Large matrices
CUDA GPU (Parallel Execution)	50-100x 🚀	Massive matrices (N > 1024)

TIME COMPLEXITY = O (N^2)

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

OPTIMIZATIONS FOR EVEN MORE SPEED :

Optimization	Benefit
Shared Memory	Reduces global memory latency
Tiling (Blocked Matrix Multiplication)	More efficient memory access
Register Usage	Reduces memory load/store operations
Tensor Cores (NVIDIA Ampere)	Faster matrix multiplication for AI/ML

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

FOR SMALL MATRICES, WHY CUDA APPROACH IS NOT SUITABLE ?

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

FOR SMALL MATRICES, WHY CUDA APPROACH IS NOT SUITABLE ?

1. ALLOCATION OF MEMORY, BACK AND FORTH DATA TRANSFER, KERNEL GENERATION IN CUDA TAKES TIME.
2. THIS MAKES THE TIME DIFFERENCE BETWEEN CUDA APPROACH AND NAIVE APPROACH NEGLIGIBLE
3. INTERESTINGLY, EIGEN APPROACH OF C++ IS FAR BETTER THAN CUDA FOR SMALL MATRICES.

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

FOR SMALL MATRICES, WHY CUDA APPROACH IS NOT SUITABLE ?

1. ALLOCATION OF MEMORY, BACK AND FORTH DATA TRANSFER, KERNEL GENERATION IN CUDA TAKES TIME.
2. THIS MAKES THE TIME DIFFERENCE BETWEEN CUDA APPROACH AND NAIVE APPROACH NEGLIGIBLE
3. INTERESTINGLY, EIGEN APPROACH OF C++ IS FAR BETTER THAN CUDA FOR SMALL MATRICES.

BUT,

CUDA (COMPUTE UNIFIED DEVICE ARCHITECTURE)

FOR SMALL MATRICES, WHY CUDA APPROACH IS NOT SUITABLE ?

1. ALLOCATION OF MEMORY, BACK AND FORTH DATA TRANSFER, KERNEL GENERATION IN CUDA TAKES TIME.
2. THIS MAKES THE TIME DIFFERENCE BETWEEN CUDA APPROACH AND NAIVE APPROACH NEGLIGIBLE
3. INTERESTINGLY, EIGEN APPROACH OF C++ IS FAR BETTER THAN CUDA FOR SMALL MATRICES.

BUT,

FOR LARGE MATRICES, DUE TO O (N^2), CUDA APPROACH IS MUCH - MUCH BETTER THAN NAIVE OR OPTIMISED APPROACH.

THANKS FOR YOUR TIME

THANKS FOR YOUR TIME

SIGNNING OFF : YUG BARGAWAY