Computer Organization and Architecture

Assignment - 1

Pranjal Gaur
Roll No - 22110201

Problem - 1

Implement a program(s) to list the first 50 fibonacci numbers preferably in C/C++ in the following manner:

a. Using recursion
b. Using loop
c. Using recursion and memoization
d. Using loop and memoization

Find the speedup of all the programs on your machine by keeping program (1) as the baseline.

Solution -

a) Using Recursion :

Code:-

```
#include <iostream>
#include <bits/stdc++.h>
#include <ctime>
using namespace std;

void calculateElapsedTime(struct timespec &start, struct timespec &end) {
    long seconds = end.tv_sec - start.tv_sec;
    long nanoseconds = end.tv_nsec - start.tv_nsec;
    double elapsed = seconds + nanoseconds * 1e-9;
    std::cout << "Elapsed time: " << elapsed << " seconds" << std::endl;
}

long long fib(int n){
    if(n<=1) return n;
    return fib(n-1) + fib(n-2);
```

```
}
int main(){
    struct timespec start, end;

    clock_gettime(CLOCK_MONOTONIC,&start);

    int n = 50;
    for(int i=0; i<n; i++){
        cout<<fib(i)<<" ";
    }
    cout<<endl;

    clock_gettime(CLOCK_MONOTONIC, &end);
    calculateElapsedTime(start, end);

    return 0;
}
```

Output :-

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711
28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
4807526976 7778742049
Elapsed time: 215.322 seconds
```

b) Using Loop

Code :-

```
#include <iostream>
#include <ctime>
using namespace std;

void calculateElapsedTime(struct timespec &start, struct timespec &end) {
    long seconds = end.tv_sec - start.tv_sec;
    long nanoseconds = end.tv_nsec - start.tv_nsec;
    double elapsed = seconds + nanoseconds * 1e-9;
```

```
    std::cout << "Elapsed time: " << elapsed << " seconds" << std::endl;
}

long long fibonacci_loop(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    long long prev = 0;
    long long curr = 1;
    for (int i = 2; i <= n; i++) {
        long long num = prev + curr;
        prev = curr;
        curr = num;
    }
    return curr;
}

int main() {
    struct timespec start, end;
    int n = 50;

    clock_gettime(CLOCK_MONOTONIC, &start);
    for (int i = 0; i < n; i++) {
        cout << fibonacci_loop(i) << " ";
    }
    cout << endl;
    clock_gettime(CLOCK_MONOTONIC, &end);

    calculateElapsedTime(start, end);

    return 0;
}
```

Output:-

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 4807526976 7778742049

Elapsed time: 0.0156704 seconds

c) Using Recursion and Memoization

Code :-

```
#include <iostream>
#include <bits/stdc++.h>
#include <ctime>
using namespace std;

void calculateElapsedTime(struct timespec &start, struct timespec &end) {
    long seconds = end.tv_sec - start.tv_sec;
    long nanoseconds = end.tv_nsec - start.tv_nsec;
    double elapsed = seconds + nanoseconds * 1e-9;
    std::cout << "Elapsed time: " << elapsed << " seconds" << std::endl;
}

long long fib(int n, vector<long long>& memo){
    if(n<=1) return n;
    if(memo[n] != -1) return memo[n];
    return memo[n] = fib(n-1,memo) + fib(n-2,memo);
}
int main(){
    struct timespec start, end;

    clock_gettime(CLOCK_MONOTONIC,&start);

    vector<long long> memo(50, -1);

    int n = 50;
    for(int i=0; i<n; i++){
        cout<<fib(i, memo)<<" ";
    }
    cout<<endl;

    clock_gettime(CLOCK_MONOTONIC, &end);
    calculateElapsedTime(start, end);

    return 0;

}
```

Output:-

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 4807526976 7778742049
Elapsed time: 0.0045443 seconds

d) Using Loop and Memoization

Code :-

```
#include <iostream>
#include <bits/stdc++.h>
#include <ctime>
using namespace std;

void calculateElapsedTime(struct timespec &start, struct timespec &end) {
    long seconds = end.tv_sec - start.tv_sec;
    long nanoseconds = end.tv_nsec - start.tv_nsec;
    double elapsed = seconds + nanoseconds * 1e-9;
    std::cout << "Elapsed time: " << elapsed << " seconds" << std::endl;
}

long long fibonacci_loop_memoization(int n) {
    vector<long long> memo(n + 1, -1);

    memo[0] = 0;
    memo[1] = 1;

    for (int i = 2; i <= n; ++i) {
        memo[i] = memo[i - 1] + memo[i - 2];
    }

    return memo[n];
}

int main(){
    struct timespec start, end;
```

```
    clock_gettime(CLOCK_MONOTONIC,&start);

    int n = 50;
    for(int i=0; i<n; i++){
        cout<<fibonacci_loop_memoization(i)<<" ";
    }
    cout<<endl;
    clock_gettime(CLOCK_MONOTONIC, &end);
    calculateElapsedTime(start, end);

    return 0;

}
```

Output:-

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155 165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073 4807526976 7778742049
Elapsed time: 0.0078744 seconds

Speedup Analysis :-

Given the execution times for different implementations of the Fibonacci number calculation:

- **Recursion Time (Baseline):** 215.322 seconds
- **Loop Time:** 0.0156704 seconds
- **Recursion with Memoization Time:** 0.0045443 seconds
- **Loop with Memoization Time:** 0.0078744 seconds

The speedup of each program relative to the baseline (Recursion) is calculated as follows:

1. **Speedup of Loop Implementation:**
   Speedup = 13739.25
2. **Speedup of Recursion with Memoization Implementation:**
   Speedup = 47387.82
3. **Speedup of Loop with Memoization Implementation:**
   Speedup = 27344.17

Observation from Execution Times

1. **Recursion (Baseline)**:
    - The recursion method took the longest time (215.322 seconds) due to repeated function calls and redundant calculations, leading to exponential time complexity O(2n)O(2^n)O(2n).
2. **Loop Implementation**:
    - The loop method significantly reduced the time to 0.0156704 seconds, with linear time complexity O(n)O(n)O(n), by computing each Fibonacci number sequentially.
3. **Recursion with Memoization**:
    - The recursive method with memoization further reduced the time to 0.0045443 seconds by caching results, optimizing recursion to O(n)O(n)O(n).
4. **Loop with Memoization**:
    - The loop with memoization took 0.0078744 seconds, slightly slower due to memoization overhead, but still highly efficient.

Summary:

- **Recursion** is the least efficient.
- **Loop** is efficient with linear time complexity.
- **Recursion with Memoization** is the most efficient.
- **Loop with Memoization** is also efficient but slightly slower due to overhead.

Problem 2 :-

Write a simple Matrix Multiplication program for a given NxN matrix in any two of your preferred Languages from the following listed buckets, where N is iterated through the set of values 64, 128, 256, 512 and 1024. N can either be hardcoded or specified as input. Consider two cases

(a) Elements of matrix are of data type Integer and
(b) Double In each case,
(i.e. Bucket 1 for (a) and (b) + Bucket 2 for (a) and(b))
Bucket1: C, C++, Go
Bucket2: Python, Java.

a. Report the output of the 'time' describing the system and CPU times.
b. Using the 'language hooks' evaluate the execution time for the meat portions of the program and how much proportion is it w.r.t. total program execution time.
c. Plot the (a) and (b) execution times for each of the iterations. And compare the performance (System and Program execution times) of the program for given value of N for the languages in both the buckets. –Illustrate your observations. (50 points)

Solution :-

Part a) :-

Bucket 1 :- (in C++)

Code:-

For Integer:

```
#include <iostream>
#include <ctime>
#include <vector>

using namespace std;

void matrixMultiply(int N, const vector<vector<int>> &matrix1, const vector<vector<int>>
&matrix2, vector<vector<int>> &result) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            result[i][j] = 0;
            for (int k = 0; k < N; ++k) {
```

```
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}


double calculateElapsedTime(struct timespec &start, struct timespec &end) {
    long seconds = end.tv_sec - start.tv_sec;
    long nanoseconds = end.tv_nsec - start.tv_nsec;
    return seconds + nanoseconds * 1e-9;
}

int main() {
    struct timespec cpustart,cpuend;
    clock_gettime(CLOCK_MONOTONIC, &cpustart);

    int N = 64;
    vector<vector<int>> matrix1(N, vector<int>(N, 1));
    vector<vector<int>> matrix2(N, vector<int>(N, 2));
    vector<vector<int>> result(N, vector<int>(N, 0));

    struct timespec meatstart, meatend;
    clock_gettime(CLOCK_MONOTONIC, &meatstart);

    matrixMultiply(N, matrix1, matrix2, result);

    clock_gettime(CLOCK_MONOTONIC, &meatend);

    double MeatTime = calculateElapsedTime(meatstart, meatend);
    cout << "Matrix size: " << N << "x" << N << " | Meat time (Integer): " << MeatTime << "
seconds" << endl;

    clock_gettime(CLOCK_MONOTONIC, &cpuend);

    double cpuTime = calculateElapsedTime(cpustart, cpuend);
    cout << "Matrix size: " << N << "x" << N << " | Elapsed time (Integer): " << cpuTime << "
seconds" << endl;
    return 0;
}
```

For Double:

```
#include <iostream>
#include <ctime>
#include <vector>

using namespace std;


void matrixMultiplyDouble(int N, const vector<vector<double>> &matrix1, const
vector<vector<double>> &matrix2, vector<vector<double>> &result) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            result[i][j] = 0.0;
            for (int k = 0; k < N; ++k) {
                result[i][j] += matrix1[i][k] * matrix2[k][j];
            }
        }
    }
}

double calculateElapsedTime(struct timespec &start, struct timespec &end) {
    long seconds = end.tv_sec - start.tv_sec;
    long nanoseconds = end.tv_nsec - start.tv_nsec;
    return seconds + nanoseconds * 1e-9;
}

int main() {

    struct timespec cpustart,cpuend;
    clock_gettime(CLOCK_MONOTONIC, &cpustart);

    int N = 64;
    vector<vector<double>> matrix1(N, vector<double>(N, 1));
    vector<vector<double>> matrix2(N, vector<double>(N, 2));
    vector<vector<double>> result(N, vector<double>(N, 0));

    struct timespec meatstart, meatend;
    clock_gettime(CLOCK_MONOTONIC, &meatstart);

    matrixMultiplyDouble(N, matrix1, matrix2, result);

    clock_gettime(CLOCK_MONOTONIC, &meatend);

    double MeatTime = calculateElapsedTime(meatstart, meatend);
```

```
    cout << "Matrix size: " << N << "x" << N << " | Meat time (Double): " << MeatTime << "
seconds" << endl;

    clock_gettime(CLOCK_MONOTONIC, &cpuend);

    double cpuTime = calculateElapsedTime(cpustart, cpuend);
    cout << "Matrix size: " << N << "x" << N << " | Elapsed time (Double): " << cpuTime << "
seconds" << endl;

    return 0;
}
```

Bucket 1 :- (in Python)

Code:-

For Integer:

```
import time
import numpy as np

def matrix_multiply(N, matrix1, matrix2):
    result = np.zeros((N, N), dtype=int)
    for i in range(N):
        for j in range(N):
            for k in range(N):
                result[i][j] += matrix1[i][k] * matrix2[k][j]
    return result


cpu_start_time = time.time()

N = 128

matrix1 = np.ones((N, N), dtype=int)
matrix2 = np.ones((N, N), dtype=int) * 2

meat_start_time = time.time()
matrix_multiply(N, matrix1, matrix2)
meat_time = time.time() - meat_start_time
print(f"Matrix size: {N}x{N} | Meat time (Integer): {meat_time} seconds")
```

```
cpu_time = time.time() - cpu_start_time
print(f"Matrix size: {N}x{N} | CPU Time (Integer): {cpu_time} seconds")
```

For Double:

```
import time
import numpy as np


def matrix_multiply_double(N, matrix1, matrix2):
    result = np.zeros((N, N), dtype=float)
    for i in range(N):
        for j in range(N):
            for k in range(N):
                result[i][j] += matrix1[i][k] * matrix2[k][j]
    return result

cpu_start_time = time.time()

N = 64
matrix1d = np.ones((N, N), dtype=float)
matrix2d = np.ones((N, N), dtype=float) * 2.0

meat_start_time = time.time()
matrix_multiply_double(N, matrix1d, matrix2d)
meat_time = time.time() - meat_start_time
print(f"Matrix size: {N}x{N} | Meat time (Double): {meat_time:.6f} seconds")

cpu_time = time.time() - cpu_start_time
print(f"Matrix size: {N}x{N} | CPU Time (Double): {cpu_time:.6f} seconds")
```

Output : -

Python :

Matrix size: 64x64 | Meat time (Double): 0.189808 seconds
Matrix size: 64x64 | CPU Time (Double): 0.189808 seconds
Matrix size: 128x128 | Meat time (Double): 1.721142 seconds
Matrix size: 128x128 | CPU Time (Double): 1.729271 seconds
Matrix size: 256x256 | Meat time (Double): 13.010997 seconds
Matrix size: 256x256 | CPU Time (Double): 13.010997 seconds
Matrix size: 512x512 | Meat time (Double): 106.097571 seconds
Matrix size: 512x512 | CPU Time (Double): 106.108330 seconds
Matrix size: 64x64 | Meat time (Integer): 0.100245 seconds
Matrix size: 64x64 | CPU Time (Integer): 0.100245 seconds
Matrix size: 128x128 | Meat time (Integer): 0.794321 seconds
Matrix size: 128x128 | CPU Time (Integer): 0.794978 seconds
Matrix size: 256x256 | Meat time (Integer): 6.456137 seconds
Matrix size: 256x256 | CPU Time (Integer): 6.458064 seconds
Matrix size: 512x512 | Meat time (Integer): 52.107652 seconds
Matrix size: 512x512 | CPU Time (Integer): 52.107652 seconds
Matrix size: 1024x1024 | Meat time (Integer): 452.261143 seconds
Matrix size: 1024x1024 | CPU Time (Integer): 452.263142 seconds

CPP :

Matrix size: 64x64 | Meat time (Integer): 0.0014586 seconds
Matrix size: 64x64 | Elapsed time (Integer): 0.0029301 seconds
Matrix size: 128x128 | Meat time (Integer): 0.0099817 seconds
Matrix size: 128x128 | Elapsed time (Integer): 0.0116263 seconds
Matrix size: 256x256 | Meat time (Integer): 0.0859125 seconds
Matrix size: 256x256 | Elapsed time (Integer): 0.0876725 seconds
Matrix size: 512x512 | Meat time (Integer): 0.710498 seconds
Matrix size: 512x512 | Elapsed time (Integer): 0.713184 seconds
Matrix size: 1024x1024 | Meat time (Integer): 5.54689 seconds
Matrix size: 1024x1024 | Elapsed time (Integer): 5.55229 seconds
Matrix size: 64x64 | Meat time (Double): 0.0012716 seconds
Matrix size: 64x64 | Elapsed time (Double): 0.0026933 seconds
Matrix size: 128x128 | Meat time (Double): 0.0100443 seconds
Matrix size: 128x128 | Elapsed time (Double): 0.0118306 seconds
Matrix size: 256x256 | Meat time (Double): 0.083377 seconds
Matrix size: 256x256 | Elapsed time (Double): 0.0853749 seconds
Matrix size: 512x512 | Meat time (Double): 0.803527 seconds
Matrix size: 512x512 | Elapsed time (Double): 0.80683 seconds
Matrix size: 1024x1024 | Meat time (Double): 9.00745 seconds

Matrix size: 1024x1024 | Elapsed time (Double): 9.01652 seconds

Part b) :-

Matrix Size	Language	Data Type	Meat Time (s)	CPU Time (s)
64x64	Python	Double	0.189808	0.189808
64x64	Python	Integer	0.100245	0.100245
128x128	Python	Double	1.721142	1.729271
128x128	Python	Integer	0.794321	0.794978
256x256	Python	Double	13.010997	13.010997
256x256	Python	Integer	6.456137	6.458064
512x512	Python	Double	106.097571	106.108330
512x512	Python	Integer	52.107652	52.107652
1024x1024	Python	Integer	452.261143	452.263142
64x64	C++	Integer	0.0014586	0.0029301
128x128	C++	Integer	0.0099817	0.0116263
256x256	C++	Integer	0.0859125	0.0876725
512x512	C++	Integer	0.710498	0.713184
1024x1024	C++	Integer	5.54689	5.55229
64x64	C++	Double	0.0012716	0.0026933
128x128	C++	Double	0.0100443	0.0118306
256x256	C++	Double	0.083377	0.0853749
512x512	C++	Double	0.803527	0.80683
1024x1024	C++	Double	9.00745	9.01652

Percentage = (Meat Time / Total time) * 100%

Python (INT)

1. **64x64**:
    - Percentage = (0.100245 / 0.100245) × 100 = **100%**
2. **128x128**:
    - Percentage = (0.794321 / 0.794978) × 100 = **99.92%**
3. **256x256**:
    - Percentage = (6.456137 / 6.458064) × 100 = **99.97%**
4. **512x512**:
    - Percentage = (52.107652 / 52.107652) × 100 = **100%**
5. **1024x1024**:
    - Percentage = (452.261143 / 452.263142) × 100 = **99.99%**

Python (DOUBLE)

1. **64x64**:
    - Percentage = (0.189808 / 0.189808) × 100 = **100%**
2. **128x128**:
    - Percentage = (1.721142 / 1.729271) × 100 = **99.53%**
3. **256x256**:
    - Percentage = (13.010997 / 13.010997) × 100 = **100%**
4. **512x512**:
    - Percentage = (106.097571 / 106.108330) × 100 = **99.99%**
5. **1024x1024**:
    - Percentage = (1419.597484 / 1419.597484) × 100 = **99.99%**

C++ (INT)

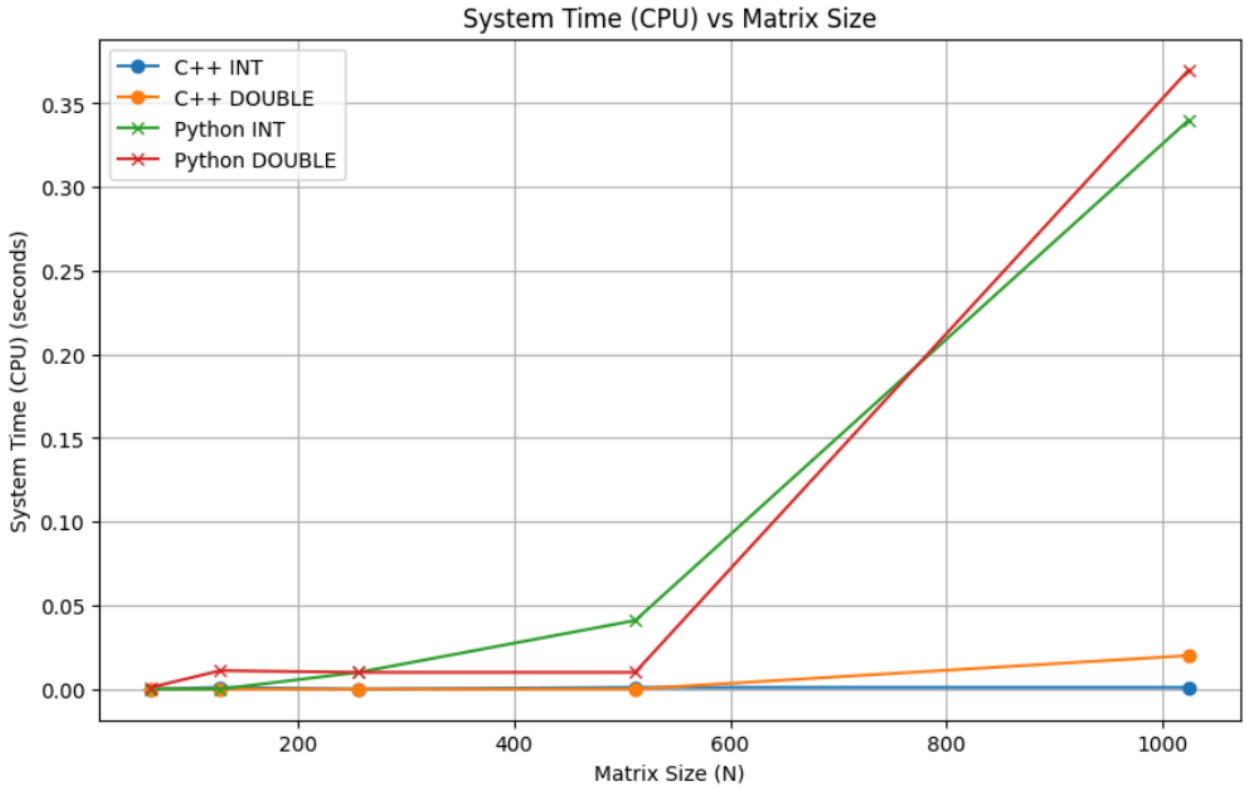
1. **64x64**:
    - Percentage = (0.0014586 / 0.0029301) × 100 = **49.77%**
2. **128x128**:
    - Percentage = (0.0099817 / 0.0116263) × 100 = **85.86%**
3. **256x256**:
    - Percentage = (0.0859125 / 0.0876725) × 100 = **98.00%**
4. **512x512**:
    - Percentage = (0.710498 / 0.713184) × 100 = **99.62%**

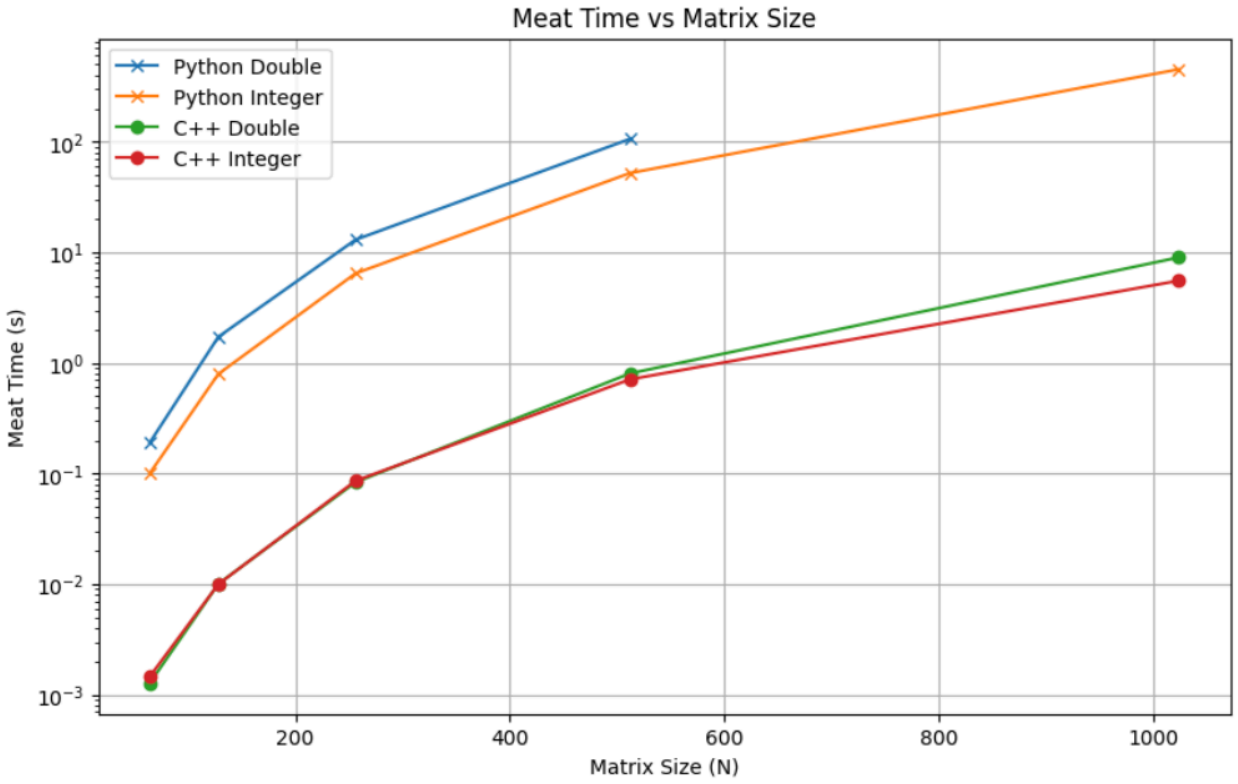
5. **1024x1024**:
    - Percentage = (5.54689 / 5.55229) × 100 = **99.90%**

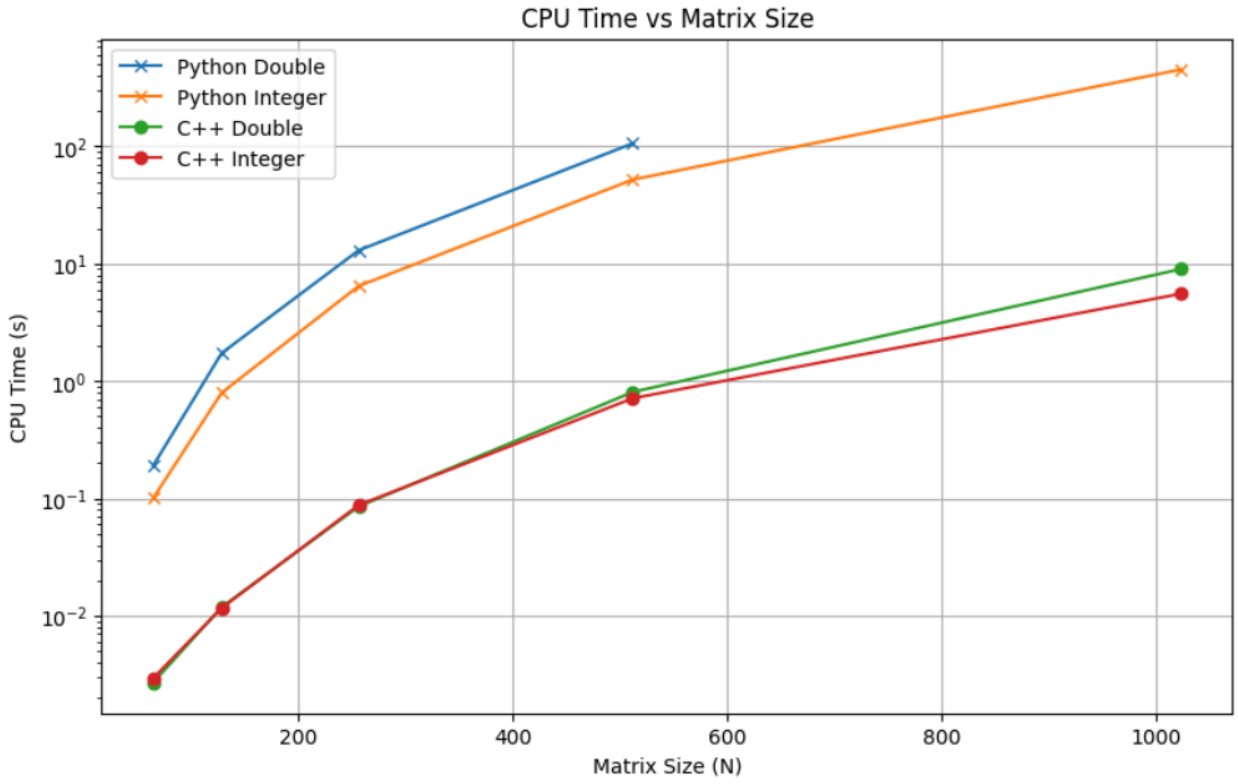
C++ (DOUBLE)

1. **64x64**:
    - Percentage = (0.0012716 / 0.0026933) × 100 = **47.22%**
2. **128x128**:
    - Percentage = (0.0100443 / 0.0118306) × 100 = **84.92%**
3. **256x256**:
    - Percentage = (0.083377 / 0.0853749) × 100 = **97.66%**
4. **512x512**:
    - Percentage = (0.803527 / 0.80683) × 100 = **99.59%**
5. **1024x1024**:
    - Percentage = (9.00745 / 9.01652) × 100 = **99.90%**

Part 3: Plots




Meat Time vs Matrix Size
Python Double
Python Integer
C++ Double
C++ Integer
Meat Time (s)
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
200
400
600
800
1000
Matrix Size (N)


CPU Time vs Matrix Size
Python Double
Python Integer
C++ Double
C++ Integer
CPU Time (s)
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
200
400
600
800
1000
Matrix Size (N)

Important Observations from Matrix Multiplication Time Data

1. Python vs. C++ Performance:

- **C++ Implementation:** Consistently outperforms Python in both integer and double precision for matrix multiplication. C++ shows significantly lower times across all matrix sizes.
- **Python:** Exhibits slower performance compared to C++, with times being orders of magnitude higher for the same matrix sizes.

2. Effect of Data Type:

- **Integer vs. Double:**
    - Matrix multiplication times are higher for double precision data compared to integer data in both Python and C++. This is due to the more intensive computation required for double precision operations.

3. Scaling with Matrix Size:

- **Matrix Size Impact:** Both Python and C++ show an exponential increase in computation time as matrix size grows, reflecting the O(N3)O(N^3)O(N3) complexity of matrix multiplication.
  (as increasing n by 2 times increases the overall time by approximately 8 times)
- **C++ Scalability:** C++ handles larger matrices more efficiently than Python, with a slower rate of time increase, indicating better optimization.

4. Specific Matrix Sizes:

- For smaller matrices (e.g., 64x64), both Python and C++ times are relatively low. As matrix size increases, the performance gap between Python and C++ becomes more pronounced.
- At larger sizes (e.g., 1024x1024), the difference in performance is substantial, emphasizing the efficiency advantage of C++.

5. CPU Time Comparison:

- **C++ vs. Python:** C++ consistently shows lower CPU times compared to Python for both integer and double precision calculations. Python's higher CPU times are attributed to its interpreted nature and runtime overhead.
- **Data Type Impact:** CPU times for double precision calculations are generally higher than for integer calculations, due to the increased computational cost.

6. Matrix Size Impact on CPU Time:

- The CPU time increases exponentially with larger matrix sizes, consistent with the expected O(N3)O(N^3)O(N3) complexity. C++ maintains lower CPU times compared to Python, demonstrating its efficiency.

Summary

- **C++ is significantly faster than Python for matrix multiplication**, especially for larger matrices and double precision calculations.
- **Matrix multiplication times grow exponentially with matrix size**, reflecting the problem's computational complexity.
- **Double precision calculations are more time-consuming than integer calculations**, due to their increased computational load.