HEAVENS' LIGHT IS OUR GUIDE

RAJSHAHI UNIVERSITY OF ENGINEERING & TECHNOLOGY

Department of Computer Science & Engineering

ALGORITHMS ANALYSIS & DESIGN SESSIONAL

Matrix Multiplication

Submitted by

Kaif Ahmed Khan

Roll: 2103163

Submitted to

A. F. M. Minhazur Rahman
Assistant Professor

December 6, 2024

Contents

List of Listings

List of Figures

1 Matrix Multiplication

1.1 Problem Statement

Implement Iterative Matrix Multiplication, Divide-and-Conquer Matrix Multiplication and Strassen's Matrix Multiplication algorithm to multiply two matrix of valid dimension. Compare the 3 algorithms based on various input size on randomly generated matrices. The comparison metric should be the execution time of each matrix multiplication algorithm.

1.2 Code

Listing 1: *matrix_multiplication.cpp*

```cpp
#include <chrono>
#include <iostream>
#include <random>
#include <vector>
using namespace std;
using namespace std::chrono;
typedef vector<vector<int>> Matrix;
Matrix generate_random_matrix(int size, int min_val = 1, int max_val = 100) {
  random_device rd;
  mt19937 gen(rd());
  uniform_int_distribution<> dis(min_val, max_val);

  Matrix matrix(size, vector<int>(size));
  for (int i = 0; i < size; ++i) {
    for (int j = 0; j < size; ++j) {
      matrix[i][j] = dis(gen);
    }
  }
  return matrix;
}

void iterative_matmul(const Matrix &A, const Matrix &B, Matrix &C, int size) {
  for (int i = 0; i < size; ++i) {
    for (int j = 0; j < size; ++j) {
      C[i][j] = 0;
      for (int k = 0; k < size; ++k) {
        C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}

void recursive_matmul(const Matrix &A, const Matrix &B, Matrix &C, int row_A,
                      int col_A, int row_B, int col_B, int row_C, int col_C,
                      int size) {
```

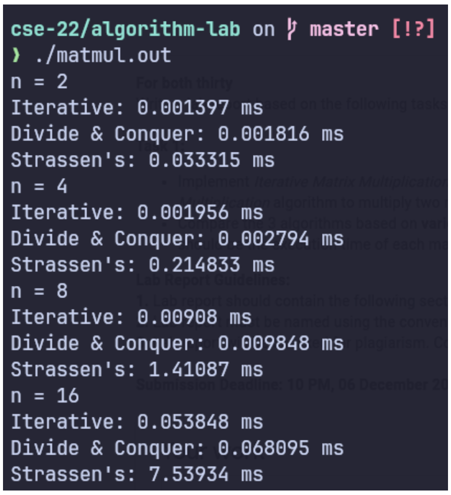
```
  if (size == 1) {
    C[row_C][col_C] += A[row_A][col_A] * B[row_B][col_B];
    return;
  }

  int new_size = size / 2;

  recursive_matmul(A, B, C, row_A, col_A, row_B, col_B, row_C, col_C, new_size);
  recursive_matmul(A, B, C, row_A, col_A + new_size, row_B + new_size, col_B,
                   row_C, col_C, new_size);

  recursive_matmul(A, B, C, row_A, col_A, row_B, col_B + new_size, row_C,
                   col_C + new_size, new_size);
  recursive_matmul(A, B, C, row_A, col_A + new_size, row_B + new_size,
                   col_B + new_size, row_C, col_C + new_size, new_size);

  recursive_matmul(A, B, C, row_A + new_size, col_A, row_B, col_B,
                   row_C + new_size, col_C, new_size);
  recursive_matmul(A, B, C, row_A + new_size, col_A + new_size,
                   row_B + new_size, col_B, row_C + new_size, col_C, new_size);

  recursive_matmul(A, B, C, row_A + new_size, col_A, row_B, col_B + new_size,
                   row_C + new_size, col_C + new_size, new_size);
  recursive_matmul(A, B, C, row_A + new_size, col_A + new_size,
                   row_B + new_size, col_B + new_size, row_C + new_size,
                   col_C + new_size, new_size);
}

Matrix add(const Matrix &A, const Matrix &B) {
  int size = A.size();
  Matrix result(size, vector<int>(size, 0));
  for (int i = 0; i < size; ++i)
    for (int j = 0; j < size; ++j)
      result[i][j] = A[i][j] + B[i][j];
  return result;
}

Matrix subtract(const Matrix &A, const Matrix &B) {
  int size = A.size();
  Matrix result(size, vector<int>(size, 0));
  for (int i = 0; i < size; ++i)
    for (int j = 0; j < size; ++j)
      result[i][j] = A[i][j] - B[i][j];
  return result;
}

Matrix strassen(const Matrix &A, const Matrix &B) {
```

```
  int size = A.size();
  Matrix C(size, vector<int>(size, 0));
  if (size == 1) {
    C[0][0] = A[0][0] * B[0][0];
    return C;
  }

  int new_size = size / 2;

  Matrix A11(new_size, vector<int>(new_size));
  Matrix A12(new_size, vector<int>(new_size));
  Matrix A21(new_size, vector<int>(new_size));
  Matrix A22(new_size, vector<int>(new_size));

  Matrix B11(new_size, vector<int>(new_size));
  Matrix B12(new_size, vector<int>(new_size));
  Matrix B21(new_size, vector<int>(new_size));
  Matrix B22(new_size, vector<int>(new_size));

  Matrix C11(new_size, vector<int>(new_size));
  Matrix C12(new_size, vector<int>(new_size));
  Matrix C21(new_size, vector<int>(new_size));
  Matrix C22(new_size, vector<int>(new_size));

  Matrix P5(new_size, vector<int>(new_size));
  Matrix P3(new_size, vector<int>(new_size));
  Matrix P1(new_size, vector<int>(new_size));
  Matrix P4(new_size, vector<int>(new_size));
  Matrix P2(new_size, vector<int>(new_size));
  Matrix P7(new_size, vector<int>(new_size));
  Matrix P6(new_size, vector<int>(new_size));

  Matrix S1(new_size, vector<int>(new_size));
  Matrix S2(new_size, vector<int>(new_size));
  Matrix S3(new_size, vector<int>(new_size));
  Matrix S4(new_size, vector<int>(new_size));
  Matrix S5(new_size, vector<int>(new_size));
  Matrix S6(new_size, vector<int>(new_size));
  Matrix S7(new_size, vector<int>(new_size));
  Matrix S8(new_size, vector<int>(new_size));
  Matrix S9(new_size, vector<int>(new_size));
  Matrix S10(new_size, vector<int>(new_size));

  for (int i = 0; i < new_size; ++i) {
    for (int j = 0; j < new_size; ++j) {
      A11[i][j] = A[i][j];
      A12[i][j] = A[i][j + new_size];
      A21[i][j] = A[i + new_size][j];
      A22[i][j] = A[i + new_size][j + new_size];

      B11[i][j] = B[i][j];
```

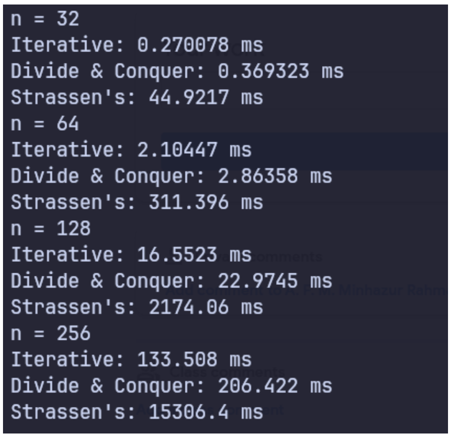
```
      B12[i][j] = B[i][j + new_size];
      B21[i][j] = B[i + new_size][j];
      B22[i][j] = B[i + new_size][j + new_size];
    }
  }
  // P1 = A11 * (B12 - B22)
  S1 = subtract(B12, B22);
  P1 = strassen(A11, S1);

  // P2 = (A11 + A12) * B22
  S2 = add(A11, A12);
  P2 = strassen(S2, B22);

  // P3 = (A21 + A22) * B11
  S3 = add(A21, A22);
  P3 = strassen(S3, B11);

  // P4 = A22 * (B21 - B11)
  S4 = subtract(B21, B11);
  P4 = strassen(A22, S4);

  // P5 = (A11 + A22) * (B11 + B22)
  S5 = add(A11, A22);
  S6 = add(B11, B22);
  P5 = strassen(S5, S6);

  // P6 = (A12 - A22) * (B21 + B22)
  S7 = subtract(A12, A22);
  S8 = add(B21, B22);
  P6 = strassen(S7, S8);

  // P7 = (A21 - A11) * (B11 + B12)
  S9 = subtract(A21, A11);
  S10 = add(B11, B12);
  P7 = strassen(S9, S10);

  C11 = subtract(add(add(P5, P4), P6), P4);
  C12 = add(P1, P2);
  C21 = add(P3, P4);
  C22 = subtract(add(add(P5, P1), P7), P3);

  // combine C11, C12, C21, C22 into C
  for (int i = 0; i < new_size; ++i) {
    for (int j = 0; j < new_size; ++j) {
      C[i][j] = C11[i][j];
      C[i][j + new_size] = C12[i][j];
      C[i + new_size][j] = C21[i][j];
      C[i + new_size][j + new_size] = C22[i][j];
    }
  }
  return C;
```

```
}

void print_matrix(const Matrix &matrix) {
  for (const auto &row : matrix) {
    for (const auto &val : row) {
      cout << val << " ";
    }
    cout << endl;
  }
}

int main() {
  for (int n = 2; n <= 256; n *= 2) {
    Matrix A = generate_random_matrix(n);
    Matrix B = generate_random_matrix(n);

    Matrix C_iterative(n, vector<int>(n, 0));
    Matrix C_recursive(n, vector<int>(n, 0));
    Matrix C_strassen(n, vector<int>(n, 0));
    cout << "n = " << n << endl;
    auto start_time = high_resolution_clock::now();
    iterative_matmul(A, B, C_iterative, n);
    auto end_time = high_resolution_clock::now();
    double time_taken =
        duration_cast<nanoseconds>(end_time - start_time).count() * 1e-6;
    cout << "Iterative: " << time_taken << " ms" << endl;

    start_time = high_resolution_clock::now();
    recursive_matmul(A, B, C_recursive, 0, 0, 0, 0, 0, 0, n);
    end_time = high_resolution_clock::now();
    time_taken =
        duration_cast<nanoseconds>(end_time - start_time).count() * 1e-6;
    cout << "Divide & Conquer: " << time_taken << " ms" << endl;

    start_time = high_resolution_clock::now();
    C_strassen = strassen(A, B);
    end_time = high_resolution_clock::now();
    time_taken =
        duration_cast<nanoseconds>(end_time - start_time).count() * 1e-6;
    cout << "Strassen's: " << time_taken << " ms" << endl;
  }
  return 0;
}
```

1.3 Output



```
cse-22/algorithm-lab on master [!?]
❯ ./matmul.out
n = 2
Iterative: 0.001397 ms
Divide & Conquer: 0.001816 ms
Strassen's: 0.033315 ms
n = 4
Iterative: 0.001956 ms
Divide & Conquer: 0.002794 ms
Strassen's: 0.214833 ms
n = 8
Iterative: 0.00908 ms
Divide & Conquer: 0.009848 ms
Strassen's: 1.41087 ms
n = 16
Iterative: 0.053848 ms
Divide & Conquer: 0.068095 ms
Strassen's: 7.53934 ms
```



```
n = 32
Iterative: 0.270078 ms
Divide & Conquer: 0.369323 ms
Strassen's: 44.9217 ms
n = 64
Iterative: 2.10447 ms
Divide & Conquer: 2.86358 ms
Strassen's: 311.396 ms
n = 128
Iterative: 16.5523 ms
Divide & Conquer: 22.9745 ms
Strassen's: 2174.06 ms
n = 256
Iterative: 133.508 ms
Divide & Conquer: 206.422 ms
Strassen's: 15306.4 ms
```

Figure 1: *Execution time for matrix multiplication*

1.4 Result

Table 1: *Comparison table for Iterative & Divide & conquer and Strassen's Matrix multiplication algorithm execution time*

Input Size	Iterative (ms)	Divide & Conquer (ms)	Strassen's (ms)
2×2	0.001397	0.001816	0.033315
4×4	0.001956	0.002794	0.214833
8×8	0.00908	0.009848	1.41087
16×16	0.053848	0.068095	7.53934
32×32	0.270078	0.369323	44.9217
64×64	2.10447	2.86358	311.396
128×128	16.5523	22.9745	2174.06
256×256	133.508	206.422	15306.4

1.5 Analysis & Discussion

The iterative algorithm demonstrates consistent performance and scales predictably with increasing input size. Its time complexity is $O(n^3)$, which aligns with theoretical expectations. As seen in the results, the runtime increases steadily, from 0.001397 ms for $2{\times}2$ matrices to 133.508 ms for $256{\times}256$ matrices. Despite its cubic complexity, the iterative approach has minimal overhead, making it an efficient choice for small to medium-sized matrices. Its straightforward implementation also contributes to its practicality in many applications.

The divide-and-conquer (recursive) algorithm shares the same $O(n^3)$ time complexity but incurs additional overhead due to recursive function calls. This overhead is noticeable for smaller matrices, where it performs slightly worse than the iterative algorithm. For example, at $2{\times}2$, the recursive approach takes 0.001816 ms, which is slower than the iterative method's 0.001397 ms. As the input size grows, the performance gap widens, with the recursive algorithm taking 206.422 ms for 256×256 matrices, compared to the iterative method's 133.508 ms. The recursion overhead thus limits its practicality for general use.

Strassen's algorithm theoretically offers better time complexity, approximately $O(n^{2.81})$, by reducing the number of multiplications required. However, in practice, it incurs significant overhead due to matrix partitioning, additional additions and subtractions, and increased recursion depth. This overhead dominates for smaller and moderately sized matrices, resulting in much slower performance compared to both iterative and recursive methods. For instance, at 64×64, Strassen's method takes 311.396 ms, compared to 2.10447 ms and 2.86358 ms for the iterative and recursive methods, respectively. For larger sizes, such as 256×256, the overhead becomes even more pronounced, with Strassen's algorithm taking 15306.4 ms, which is substantially slower than the other methods.

The results highlight the impact of algorithmic overhead. While Strassen's algorithm provides theoretical improvements, its practical utility is hindered by its computational and memory overhead. The additional memory requirements for submatrices and intermediate computations can also lead to cache inefficiencies, particularly for larger matrices. These factors explain why Strassen's algorithm underperforms compared to simpler methods in the tested range.

In conclusion, the iterative algorithm remains the most practical and efficient choice for small to medium-sized matrices due to its simplicity and low overhead. While Strassen's algorithm is theoretically advantageous, its overhead makes it unsuitable for smaller matrices, and it only becomes competitive for significantly larger input sizes. Future optimizations, such as hybrid approaches or parallelized implementations, may help mitigate Strassen's overhead and unlock its potential for practical applications involving very large matrices.