C++ Matrix Multiplication with Advanced Optimizations

This document presents a detailed, highly optimized, and performance-focused C++ code that demonstrates advanced data structures, multi-threading, parallelism, and optimized algorithms. The example illustrates a multi-threaded matrix multiplication program that uses multiple optimization techniques.

```
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>
#include <chrono>
#include <algorithm>
#include <numeric>
#include <random>
#include <functional>
#include <cassert>

// Custom matrix class to manage large matrix operations
class Matrix {
private:
    std::vector<std::vector<int>> data;
    size_t rows, cols;

public:
    // Constructor: Initializes matrix with random values
    Matrix(size_t r, size_t c) : rows(r), cols(c) {
        data.resize(rows, std::vector<int>(cols));
        // Fill with random values for simulation
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<> dis(1, 100);

        for (auto& row : data) {
            std::generate(row.begin(), row.end(), [&]() { return dis(gen); });
        }
    }
```

```
    // Getter functions
    size_t getRows() const { return rows; }
    size_t getCols() const { return cols; }
    int at(size_t row, size_t col) const { return data[row][col]; }
    int& at(size_t row, size_t col) { return data[row][col]; }

    // Matrix printing for debugging
    void print() const {
        for (const auto& row : data) {
            for (const auto& val : row) {
                std::cout << val << " ";
            }
            std::cout << "\n";
        }
    }

    // Function to perform basic addition of matrices
    static Matrix addMatrices(const Matrix& matA, const Matrix& matB) {
        assert(matA.getRows() == matB.getRows() && matA.getCols() ==
matB.getCols());
        Matrix result(matA.getRows(), matA.getCols());
        for (size_t i = 0; i < matA.getRows(); ++i) {
            for (size_t j = 0; j < matA.getCols(); ++j) {
                result.at(i, j) = matA.at(i, j) + matB.at(i, j);
            }
        }
        return result;
    }

    // Function to perform matrix subtraction
    static Matrix subtractMatrices(const Matrix& matA, const Matrix& matB) {
        assert(matA.getRows() == matB.getRows() && matA.getCols() ==
matB.getCols());
        Matrix result(matA.getRows(), matA.getCols());
        for (size_t i = 0; i < matA.getRows(); ++i) {
            for (size_t j = 0; j < matA.getCols(); ++j) {
                result.at(i, j) = matA.at(i, j) - matB.at(i, j);
            }
        }
```

```
        return result;
    }
};

// Classic matrix multiplication (single-threaded version)
Matrix multiplyMatrices(const Matrix& matA, const Matrix& matB) {
    if (matA.getCols() != matB.getRows()) {
        throw std::invalid_argument("Matrix dimensions must match for multiplication");
    }
    Matrix result(matA.getRows(), matB.getCols());
    for (size_t i = 0; i < matA.getRows(); ++i) {
        for (size_t j = 0; j < matB.getCols(); ++j) {
            int sum = 0;
            for (size_t k = 0; k < matA.getCols(); ++k) {
                sum += matA.at(i, k) * matB.at(k, j);
            }
            result.at(i, j) = sum;
        }
    }
    return result;
}

// Matrix multiplication using multiple threads
void multiplyRowByColumn(const Matrix& matA, const Matrix& matB, Matrix& result,
size_t row) {
    for (size_t j = 0; j < matB.getCols(); ++j) {
        int sum = 0;
        for (size_t k = 0; k < matA.getCols(); ++k) {
            sum += matA.at(row, k) * matB.at(k, j);
        }
        result.at(row, j) = sum;
    }
}

// Parallel matrix multiplication using threads
Matrix multiplyMatricesParallel(const Matrix& matA, const Matrix& matB) {
    if (matA.getCols() != matB.getRows()) {
        throw std::invalid_argument("Matrix dimensions must match for multiplication");
    }
    Matrix result(matA.getRows(), matB.getCols());
```

```
    std::vector<std::thread> threads;
    for (size_t i = 0; i < matA.getRows(); ++i) {
        threads.push_back(std::thread(multiplyRowByColumn, std::cref(matA),
std::cref(matB), std::ref(result), i));
    }
    for (auto& t : threads) {
        t.join();
    }
    return result;
}

// Strassen's Algorithm (Advanced matrix multiplication)
Matrix strassenMultiply(const Matrix& matA, const Matrix& matB) {
    size_t n = matA.getRows();
    size_t m = matA.getCols();
    size_t p = matB.getCols();

    if (n == 1 || m == 1 || p == 1) {
        return multiplyMatrices(matA, matB); // Base case, fall back to standard
multiplication
    }

    // Divide the matrices into submatrices
    size_t mid = n / 2;
    Matrix A11(mid, mid), A12(mid, mid), A21(mid, mid), A22(mid, mid);
    Matrix B11(mid, mid), B12(mid, mid), B21(mid, mid), B22(mid, mid);

    for (size_t i = 0; i < mid; ++i) {
        for (size_t j = 0; j < mid; ++j) {
            A11.at(i, j) = matA.at(i, j);
            A12.at(i, j) = matA.at(i, j + mid);
            A21.at(i, j) = matA.at(i + mid, j);
            A22.at(i, j) = matA.at(i + mid, j + mid);

            B11.at(i, j) = matB.at(i, j);
            B12.at(i, j) = matB.at(i, j + mid);
            B21.at(i, j) = matB.at(i + mid, j);
            B22.at(i, j) = matB.at(i + mid, j + mid);
        }
    }
```

```
    // Compute intermediate matrices
    Matrix M1 = strassenMultiply(A11, subtractMatrices(B12, B22));
    Matrix M2 = strassenMultiply(addMatrices(A11, A12), B22);
    Matrix M3 = strassenMultiply(addMatrices(A21, A22), B11);
    Matrix M4 = strassenMultiply(A22, subtractMatrices(B21, B11));
    Matrix M5 = strassenMultiply(addMatrices(A11, A22), addMatrices(B11, B22));
    Matrix M6 = strassenMultiply(subtractMatrices(A12, A22), addMatrices(B21, B22));
    Matrix M7 = strassenMultiply(subtractMatrices(A11, A21), addMatrices(B11, B12));

    // Combine the results into a final result
    Matrix result(n, p);
    for (size_t i = 0; i < mid; ++i) {
        for (size_t j = 0; j < mid; ++j) {
            result.at(i, j) = M5.at(i, j) + M4.at(i, j) - M2.at(i, j) + M6.at(i, j);
            result.at(i, j + mid) = M1.at(i, j) + M2.at(i, j);
            result.at(i + mid, j) = M3.at(i, j) + M4.at(i, j);
            result.at(i + mid, j + mid) = M5.at(i, j) + M1.at(i, j) - M3.at(i, j) - M7.at(i, j);
        }
    }
    return result;
}

// Function to benchmark execution time
template <typename Func, typename... Args>
double benchmarkExecutionTime(Func&& func, Args&&... args) {
    auto start = std::chrono::high_resolution_clock::now();
    func(std::forward<Args>(args)...);
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> duration = end - start;
    return duration.count();
}

int main() {
    size_t rows = 512, cols = 512;

    // Create random matrices
    Matrix matA(rows, cols);
    Matrix matB(cols, rows);
```

```
    // Benchmarking the matrix multiplication methods
    std::cout << "Starting standard matrix multiplication...\n";
    double timeTaken = benchmarkExecutionTime(multiplyMatrices, std::cref(matA),
std::cref(matB));
    std::cout << "Standard matrix multiplication took: " << timeTaken << " seconds.\n";

    std::cout << "\nStarting parallel matrix multiplication...\n";
    timeTaken = benchmarkExecutionTime(multiplyMatricesParallel, std::cref(matA),
std::cref(matB));
    std::cout << "Parallel matrix multiplication took: " << timeTaken << " seconds.\n";

    std::cout << "\nStarting Strassen's algorithm matrix multiplication...\n";
    timeTaken = benchmarkExecutionTime(strassenMultiply, std::cref(matA),
std::cref(matB));
    std::cout << "Strassen's algorithm matrix multiplication took: " << timeTaken << "
seconds.\n";

    return 0;
}
```