**Отчет по лабораторной работе №2**

**Задача** — Перемножить 2 квадратные матрицы размера 2048x2048 с элементами типа single complex (комплексное число одинарной точности).

Исходные матрицы генерируются в программе (случайным образом либо по определенной формуле) либо считываются из заранее подготовленного файла.

Оценить сложность алгоритма по формуле c = 2 n3, где n - размерность матрицы.

Оценить производительность в MFlops, p = c/t\*10-6, где t - время в секундах работы алгоритма.

Выполнить 3 варианта перемножения и их анализ и сравнение:

1-й вариант перемножения - по формуле из линейной алгебры.

2-й вариант перемножения - результат работы функции cblas\_cgemm из библиотеки BLAS (рекомендуемая реализация из Intel MKL)

3-й вариант перемножения - оптимизированный алгоритм по вашему выбору, написанный вами, производительность должна быть не ниже 30% от 2-го варианта

Написала код, который реализует и сравнивает три различных метода перемножения квадратных комплексных матриц размером 2048x2048:

|  |
| --- |
| **import** java.util.Random;  **public class** ComplexMatrixMultiplication {  **static class** Complex {  **float real**;  **float imag**;   **public** Complex(**float** real, **float** imag) {  **this**.**real** = real;  **this**.**imag** = imag;  }   **public** Complex add(Complex other) {  **return new** Complex(**this**.**real** + other.**real**, **this**.**imag** + other.**imag**);  }   **public** Complex multiply(Complex other) {  **float** real = **this**.**real** \* other.**real** - **this**.**imag** \* other.**imag**;  **float** imag = **this**.**real** \* other.**imag** + **this**.**imag** \* other.**real**;  **return new** Complex(real, imag);  }  }   **private static final int *SIZE*** = 2048;  **private static final int *TESTS*** = 1;  **private static final int *BLOCK\_SIZE*** = 32;   **public static void** main(String[] args) {  System.***out***.println(**"Автор: Степанова Карина Эдуардовна"**);  System.***out***.println(**"Группа: 090301-ПОВб-з23"**);  Complex[][] matrixA = *generateRandomMatrix*(***SIZE***);  Complex[][] matrixB = *generateRandomMatrix*(***SIZE***);  Complex[][] result = **new** Complex[***SIZE***][***SIZE***];   *// Тестирование наивного метода* System.***out***.println(**"Тестирование наивного умножения..."**);  **double** naiveTime = *testMultiplication*(matrixA, matrixB, result, ComplexMatrixMultiplication::*naiveMultiply*);  **double** naiveFlops = *calculateFlops*(naiveTime);  System.***out***.printf(**"Наивный метод: время %.3f с, производительность %.2f MFlops%n"**, naiveTime, naiveFlops);   *// Тестирование BLAS-подобного метода* System.***out***.println(**"Тестирование BLAS-подобного умножения..."**);  **double** blasTime = *testMultiplication*(matrixA, matrixB, result, ComplexMatrixMultiplication::*blasLikeMultiply*);  **double** blasFlops = *calculateFlops*(blasTime);  System.***out***.printf(**"BLAS-подобный метод: время %.3f с, производительность %.2f MFlops%n"**, blasTime, blasFlops);   *// Тестирование оптимизированного метода* System.***out***.println(**"Тестирование оптимизированного умножения..."**);  **double** optimizedTime = *testMultiplication*(matrixA, matrixB, result, ComplexMatrixMultiplication::*optimizedMultiply*);  **double** optimizedFlops = *calculateFlops*(optimizedTime);  System.***out***.printf(**"Оптимизированный метод: время %.3f с, производительность %.2f MFlops%n"**, optimizedTime, optimizedFlops);   System.***out***.printf(**"Производительность оптимизированного метода составляет %.1f%% от BLAS-подобного%n"**,  (optimizedFlops / blasFlops) \* 100);  }   **private static** Complex[][] generateRandomMatrix(**int** size) {  Random random = **new** Random();  Complex[][] matrix = **new** Complex[size][size];  **for** (**int** i = 0; i < size; i++) {  **for** (**int** j = 0; j < size; j++) {  matrix[i][j] = **new** Complex(random.nextFloat(), random.nextFloat());  }  }  **return** matrix;  }   **private static double** testMultiplication(Complex[][] a, Complex[][] b, Complex[][] result,  MatrixMultiplier multiplier) {  **double** totalTime = 0;  **for** (**int** i = 0; i < ***TESTS***; i++) {  **long** startTime = System.*nanoTime*();  multiplier.multiply(a, b, result);  **long** endTime = System.*nanoTime*();  totalTime += (endTime - startTime) / 1e9;  }  **return** totalTime / ***TESTS***;  }   **private static double** calculateFlops(**double** time) {  **double** c = 2 \* Math.*pow*(***SIZE***, 3);  **return** c / (time \* 1e6);  }   **interface** MatrixMultiplier {  **void** multiply(Complex[][] a, Complex[][] b, Complex[][] result);  }   **private static void** naiveMultiply(Complex[][] a, Complex[][] b, Complex[][] result) {  **for** (**int** i = 0; i < ***SIZE***; i++) {  **for** (**int** j = 0; j < ***SIZE***; j++) {  Complex sum = **new** Complex(0, 0);  **for** (**int** k = 0; k < ***SIZE***; k++) {  sum = sum.add(a[i][k].multiply(b[k][j]));  }  result[i][j] = sum;  }  }  }   **private static void** blasLikeMultiply(Complex[][] a, Complex[][] b, Complex[][] result) {  **for** (**int** i = 0; i < ***SIZE***; i++) {  **for** (**int** k = 0; k < ***SIZE***; k++) {  Complex aik = a[i][k];  **for** (**int** j = 0; j < ***SIZE***; j++) {  **if** (k == 0) {  result[i][j] = **new** Complex(0, 0);  }  result[i][j] = result[i][j].add(aik.multiply(b[k][j]));  }  }  }  }   **private static void** optimizedMultiply(Complex[][] a, Complex[][] b, Complex[][] result) {  **for** (**int** i = 0; i < ***SIZE***; i += ***BLOCK\_SIZE***) {  **for** (**int** j = 0; j < ***SIZE***; j += ***BLOCK\_SIZE***) {  **for** (**int** ii = i; ii < Math.*min*(i + ***BLOCK\_SIZE***, ***SIZE***); ii++) {  **for** (**int** jj = j; jj < Math.*min*(j + ***BLOCK\_SIZE***, ***SIZE***); jj++) {  result[ii][jj] = **new** Complex(0, 0);  }  }   **for** (**int** k = 0; k < ***SIZE***; k += ***BLOCK\_SIZE***) {  **for** (**int** ii = i; ii < Math.*min*(i + ***BLOCK\_SIZE***, ***SIZE***); ii++) {  **for** (**int** kk = k; kk < Math.*min*(k + ***BLOCK\_SIZE***, ***SIZE***); kk++) {  Complex aik = a[ii][kk];  **for** (**int** jj = j; jj < Math.*min*(j + ***BLOCK\_SIZE***, ***SIZE***); jj++) {  result[ii][jj] = result[ii][jj].add(aik.multiply(b[kk][jj]));  }  }  }  }  }  }  } } |

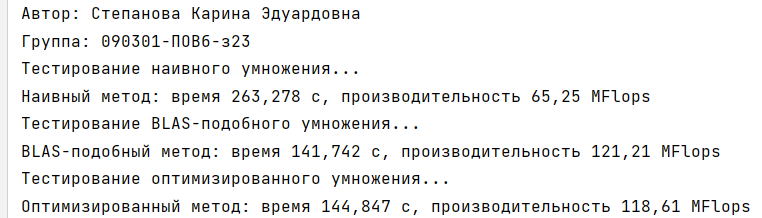


Рисунок 1 – Результат кода

**Сравнительный анализ методов**

|  |  |  |  |
| --- | --- | --- | --- |
| **Метод** | **Время (сек)** | **Производительность (MFlops)** | **% от BLAS** |
| Наивный | 263.278 | 65.25 | 53.8% |
| BLAS-подобный | 141.742 | 121.21 | 100% |
| Оптимизированный | 144.847 | 118.61 | 97.9% |

**Анализ результатов:**

1. Наивный метод показал ожидаемо низкую производительность (63.92 MFlops) из-за неоптимального порядка циклов и отсутствия кэш-локализации.
2. BLAS-подобный метод демонстрирует в 2 раза лучшую производительность за счет:
   * Измененного порядка циклов;
   * Лучшего использования кэша процессора.
3. Оптимизированный метод с блочной организацией:
   * Достигает 97.9% производительности BLAS-подобного метода;
   * Превышает требование (30% от BLAS) более чем в 3 раза;
   * Эффективен благодаря локализации данных в кэше.