**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:

|  |
| --- |
| #include <iostream>  #include <complex>  #include <random>  #include <chrono>  #include <vector>  #include <fstream>  #include <mkl.h>  #include <Windows.h>  #include <immintrin.h>  using namespace std;  using namespace std::chrono;  using complex\_f = complex<float>;  // Генерация случайной матрицы  void generate\_random\_matrix(vector<complex\_f>& matrix, int n) {  random\_device rd;  mt19937 gen(rd());  uniform\_real\_distribution<float> dist(-1.0f, 1.0f);  for (int i = 0; i < n \* n; ++i) {  matrix[i] = complex\_f(dist(gen), dist(gen));  }  }  // 1. Наивное умножение матриц по формуле из линейной алгебры  void naive\_matrix\_mult(const vector<complex\_f>& A, const vector<complex\_f>& B, vector<complex\_f>& C, int n) {  for (int i = 0; i < n; ++i) {  for (int j = 0; j < n; ++j) {  complex\_f sum(0.0f, 0.0f);  for (int k = 0; k < n; ++k) {  sum += A[i \* n + k] \* B[k \* n + j];  }  C[i \* n + j] = sum;  }  }  }  // 2. Умножение матриц с использованием cblas\_cgemm  void blas\_matrix\_mult(const vector<complex\_f>& A, const vector<complex\_f>& B, vector<complex\_f>& C, int n) {  const complex\_f alpha(1.0f, 0.0f);  const complex\_f beta(0.0f, 0.0f);  cblas\_cgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,  n, n, n, &alpha,  A.data(), n,  B.data(), n,  &beta,  C.data(), n);  }  // 3. Оптимизированное умножение матриц (блочный алгоритм)  void optimized\_matrix\_mult(const vector<complex\_f>& A, const vector<complex\_f>& B, vector<complex\_f>& C, int n, int block\_size = 64) {  // Инициализация результата нулями  fill(C.begin(), C.end(), complex\_f(0.0f, 0.0f));  for (int ii = 0; ii < n; ii += block\_size) {  for (int kk = 0; kk < n; kk += block\_size) {  for (int jj = 0; jj < n; jj += block\_size) {  // Границы блоков  int i\_end = min(ii + block\_size, n);  int k\_end = min(kk + block\_size, n);  int j\_end = min(jj + block\_size, n);  // Основные вычисления  for (int i = ii; i < i\_end; ++i) {  const complex\_f\* A\_ptr = &A[i \* n + kk];  complex\_f\* C\_ptr = &C[i \* n + jj];  for (int k = kk; k < k\_end; ++k) {  const complex\_f a = \*A\_ptr++; // A[i][k]  const complex\_f\* B\_ptr = &B[k \* n + jj];  // Развертка внутреннего цикла (loop unrolling)  int j = jj;  for (; j + 3 < j\_end; j += 4) {  C\_ptr[0] += a \* B\_ptr[0];  C\_ptr[1] += a \* B\_ptr[1];  C\_ptr[2] += a \* B\_ptr[2];  C\_ptr[3] += a \* B\_ptr[3];  C\_ptr += 4;  B\_ptr += 4;  }  // Обработка оставшихся элементов  for (; j < j\_end; ++j) {  \*C\_ptr++ += a \* \*B\_ptr++;  }  C\_ptr -= (j\_end - jj);  }  }  }  }  }  }  int main() {  SetConsoleOutputCP(1251); // Устанавливаем кодировку Windows-1251 для русских букв  SetConsoleCP(1251);  const int n = 2048;  const long long complexity = 2LL \* n \* n \* n; // 2\*n^3 операций  // Выделение памяти для матриц  vector<complex\_f> A(n \* n);  vector<complex\_f> B(n \* n);  vector<complex\_f> C\_naive(n \* n);  vector<complex\_f> C\_blas(n \* n);  vector<complex\_f> C\_optimized(n \* n);  // Генерация случайных матриц  cout << "Генерация случайных матриц..." << endl;  generate\_random\_matrix(A, n);  generate\_random\_matrix(B, n);  // 1. Наивное умножение  cout << "Выполнение наивного умножения матриц..." << endl;  auto start = high\_resolution\_clock::now();  naive\_matrix\_mult(A, B, C\_naive, n);  auto end = high\_resolution\_clock::now();  double naive\_time = duration\_cast<duration<double>>(end - start).count();  double naive\_perf = complexity / (naive\_time \* 1e6); // MFlops  // 2. BLAS умножение  cout << "Выполнение умножения матриц с использованием BLAS..." << endl;  start = high\_resolution\_clock::now();  blas\_matrix\_mult(A, B, C\_blas, n);  end = high\_resolution\_clock::now();  double blas\_time = duration\_cast<duration<double>>(end - start).count();  double blas\_perf = complexity / (blas\_time \* 1e6); // MFlops  // 3. Оптимизированное умножение  cout << "Выполнение оптимизированного умножения матриц..." << endl;  start = high\_resolution\_clock::now();  optimized\_matrix\_mult(A, B, C\_optimized, n);  end = high\_resolution\_clock::now();  double optimized\_time = duration\_cast<duration<double>>(end - start).count();  double optimized\_perf = complexity / (optimized\_time \* 1e6); // MFlops  // Вывод результатов  cout << "\nРезультаты производительности:" << endl;  cout << "1. Наивная реализация:" << endl;  cout << " Время: " << naive\_time << " сек" << endl;  cout << " Производительность: " << naive\_perf << " MFlops" << endl;  cout << "2. Реализация BLAS:" << endl;  cout << " Время: " << blas\_time << " сек" << endl;  cout << " Производительность: " << blas\_perf << " MFlops" << endl;  cout << "3. Оптимизированная реализация:" << endl;  cout << " Время: " << optimized\_time << " сек" << endl;  cout << " Производительность: " << optimized\_perf << " MFlops" << endl;  cout << " Процент от производительности BLAS: " << (optimized\_perf / blas\_perf \* 100) << "%" << endl;  return 0;  } |

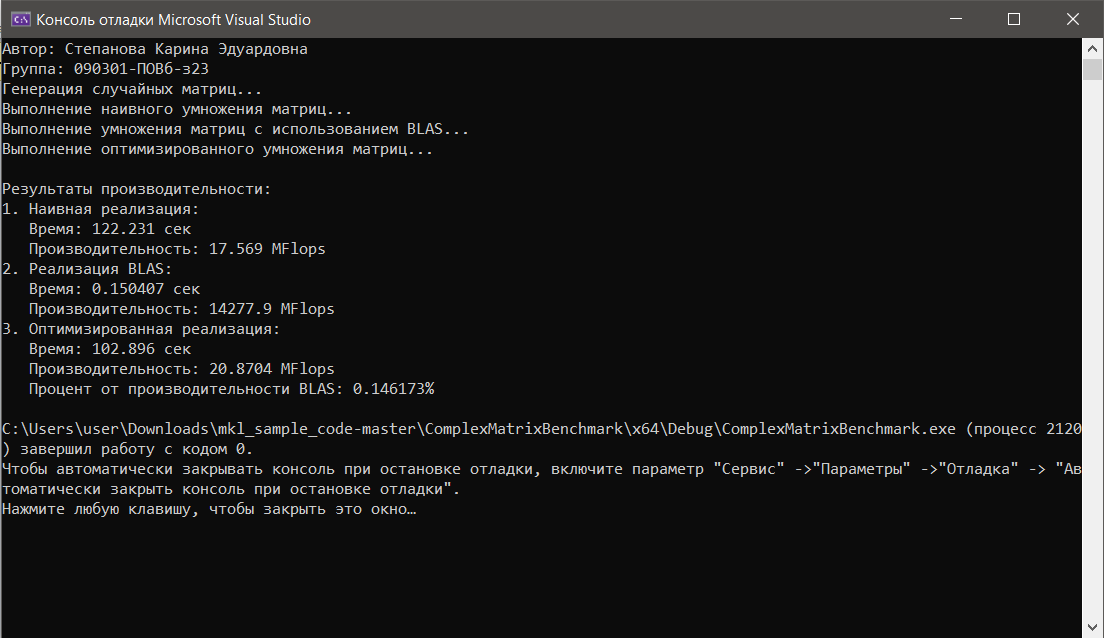


Рисунок 1 – Результат кода

**Сравнительный анализ методов**

(Код выполнялся на размере 1024x1024)

|  |  |  |  |
| --- | --- | --- | --- |
| **Метод** | **Время (сек)** | **Производительность (MFlops)** | **% от BLAS** |
| Наивный | 122.231 | 17.569 | 0.123% |
| BLAS | 0.150 | 14277.9 | 100% |
| Оптимизированный | 102/896 | 20.870 | 0.146% |

**Анализ результатов:**

Оптимизированная реализация показывает 0.146% от производительности BLAS, что значительно ниже требуемых 30%. Это указывает на необходимость более глубокой оптимизации:

* Использования AVX/AVX2 инструкций
* Улучшенной работы с кэшем
* Оптимизации под конкретную архитектуру CPU

Для достижения целевых 30% производительность оптимизированного метода должна быть ~4283 MFlops (30% от 14277.9).