**ЛАБОРАТОРНАЯ РАБОТА №2**

**Цель работы:** Научиться создавать поиск по определенным условиям.

**Задание:** Перемножить 2 квадратные матрицы размера 1024x1024 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.

**Описание работы программы:**

Программа автоматически создаёт 2 матрицы и постепенно перемнажает их тремя разными способами. Получается три матрицы, которые сравниваются между собой и выводятся 5 случайных элементов, каждой из этих 3 матриц.

**Листинг программы через простой массив:**

#include <iostream>

#include <vector>

#include <complex>

#include <random>

#include <chrono>

#include <iomanip>

#include <cblas.h>

#ifdef \_OPENMP

#include <omp.h>

#endif

using namespace std;

using Complex = std::complex<float>;

const int sz = 1024;

const int a = -1;

const int b = 5;

constexpr int BSIZE = 32;

void mat1(const vector<complex<float>>& am, const vector<complex<float>>& bm, vector<complex<float>>& cm) {

for (int i = 0; i < sz; i++) {

for (int j = 0; j < sz; j++) {

complex<float> js(0.0, 0.0);

for (int k = 0; k < sz; k++) {

js += am[i \* sz + k] \* bm[k \* sz + j];

}

cm[i \* sz + j] = js;

}

}

}

void mat2(const vector<complex<float>>& as, const vector<complex<float>>& bs, vector<complex<float>>& cs) {

CBLAS\_LAYOUT layout = CblasRowMajor;

CBLAS\_TRANSPOSE transa = CblasNoTrans;

CBLAS\_TRANSPOSE transb = CblasNoTrans;

const int m = sz;

const int n = sz;

const int k = sz;

complex<float> alpha(1.0f, 0.0f);

complex<float> beta(0.0f, 0.0f);

const int lda = sz;

const int ldb = sz;

const int ldc = sz;

cblas\_cgemm(layout, transa, transb, m, n, k, &alpha, as.data(), lda, bs.data(), ldb, &beta, cs.data(), ldc);

}

void mat3(const std::vector<Complex>& A,

const std::vector<Complex>& B,

std::vector<Complex>& C) {

#pragma omp parallel for if(sz >= 256) schedule(static)

for (int i = 0; i < sz \* sz; ++i) {

C[i] = Complex(0.0f, 0.0f);

}

#pragma omp parallel for collapse(2) schedule(static)

for (int iBlock = 0; iBlock < sz; iBlock += BSIZE) {

for (int jBlock = 0; jBlock < sz; jBlock += BSIZE) {

for (int kBlock = 0; kBlock < sz; kBlock += BSIZE) {

int iMax = std::min(iBlock + BSIZE, sz);

int jMax = std::min(jBlock + BSIZE, sz);

int kMax = std::min(kBlock + BSIZE, sz);

for (int i = iBlock; i < iMax; ++i) {

for (int k = kBlock; k < kMax; ++k) {

Complex a\_val = A[i \* sz + k];

int a\_pos = i \* sz + k;

for (int j = jBlock; j < jMax; ++j) {

int b\_pos = k \* sz + j;

int c\_pos = i \* sz + j;

C[c\_pos] += a\_val \* B[b\_pos];

}

}

}

}

}

}

}

int main() {

system("chcp 1251");

system("cls");

cout << "Выполнил: Карпенко Денис Иванович\nГруппа: 020303-АИСа-24о\n";

random\_device rd;

mt19937 gen(rd());

uniform\_real\_distribution<float> dis(a, a + b);

vector<complex<float>> am(sz \* sz);

vector<complex<float>> bm(sz \* sz);

vector<complex<float>> c1(sz \* sz, complex<float>(0.0f, 0.0f));

vector<complex<float>> c2(sz \* sz, complex<float>(0.0f, 0.0f));

vector<complex<float>> c3(sz \* sz, complex<float>(0.0f, 0.0f));

for (int i = 0; i < sz \* sz; ++i) {

am[i] = complex<float>(dis(gen), dis(gen));

bm[i] = complex<float>(dis(gen), dis(gen));

}

cout << "Вычисление первой матрицы C1:\n";

auto start1 = chrono::high\_resolution\_clock::now();

mat1(am, bm, c1);

auto end1 = chrono::high\_resolution\_clock::now();

chrono::duration<double> el\_s1 = end1 - start1;

double t = el\_s1.count();

double cty = 2.0 \* pow(sz, 3);

double mfl1 = cty / t / 1e6;

cout << fixed << setprecision(2);

cout << "Размер массива cm: " << (sizeof(complex<float>) \* sz \* sz) / 1024 << " KB" << endl;

cout << "Сложность алгоритма: " << cty << endl;

cout << "Время выполнения: " << t << " секунд" << endl;

cout << "Производительность: " << mfl1 << " MFlops" << endl;

cout << "\nВычисление первой матрицы C2:\n";

auto start2 = chrono::high\_resolution\_clock::now();

mat2(am, bm, c2);

auto end2 = chrono::high\_resolution\_clock::now();

chrono::duration<double> el\_s2 = end2 - start2;

double ti = el\_s2.count();

double ct = 2.0 \* pow(sz, 3);

double mfl2 = ct / ti / 1e6;

cout << fixed << setprecision(2);

cout << "Размер массива cm: " << (sizeof(complex<float>) \* sz \* sz) / 1024 << " KB" << endl;

cout << "Сложность алгоритма: " << ct << endl;

cout << "Время выполнения: " << ti << " секунд" << endl;

cout << "Производительность: " << mfl2 << " MFlops" << endl;

cout << "\nВычисление первой матрицы C3:\n";

auto start3 = chrono::high\_resolution\_clock::now();

mat3(am, bm, c3);

auto end3 = chrono::high\_resolution\_clock::now();

chrono::duration<double> el\_s3 = end3 - start3;

double tt = el\_s3.count();

double ctt = 2.0 \* pow(sz, 3);

double mfl3 = ctt / tt / 1e6;

cout << fixed << setprecision(2);

cout << "Размер массива cm: " << (sizeof(complex<float>) \* sz \* sz) / 1024 << " KB" << endl;

cout << "Сложность алгоритма: " << ctt << endl;

cout << "Время выполнения: " << tt << " секунд" << endl;

cout << "Производительность: " << mfl3 << " MFlops" << endl;

cout << "\nПроверка матриц 5-ю случайными элементами:\n";

for (int i = 0; i < 5; ++i) {

int row = rand() % sz;

int col = rand() % sz;

cout << "C1[" << row << "][" << col << "] = " << c1[row \* sz + col] << " ";

if (abs(c2[row \* sz + col].real() - c1[row \* sz + col].real()) < 1e-2 &&

abs(c2[row \* sz + col].imag() - c1[row \* sz + col].imag()) < 1e-2) {

cout << "correct\n";

}

else {

cout << "false\n";

}

cout << "C2[" << row << "][" << col << "] = " << c2[row \* sz + col] << endl;

cout << "C3[" << row << "][" << col << "] = " << c3[row \* sz + col] << " ";

if (abs(c2[row \* sz + col].real() - c3[row \* sz + col].real()) < 1e-2 &&

abs(c2[row \* sz + col].imag() - c3[row \* sz + col].imag()) < 1e-2) {

cout << "correct\n";

}

else {

cout << "false\n";

}

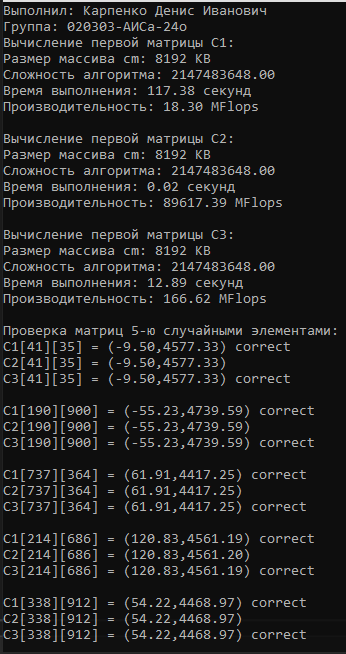
cout << endl;

}

return 0;

}

**Пример работы программы:**



**Вывод:** Были получены знания по тому как можно оптимизировать свой код для больших объёмов данных.