ЛАБОРАТОРНАЯ РАБОТА №2

ОПТИМИЗАЦИЯ РАБОТЫ С БОЛЬШИМИ ДАННЫМИ (НА ПРИМЕРЕ ПЕРЕМНОЖЕНИЯ МАТРИЦ)

ЗАДАНИЕ

Перемножить 2 квадратные матрицы размера 4096x4096 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 $c = 2\ n^3$, 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 вариант

ХОД РАБОТЫ

Single complex (комплексное число одинарной точности), оно же std::complex<float> в C++, занимает в памяти ровно 8 байт – 4 байта на вещественную часть (Re) и ещё 4 на мнимую (Im).

Классический алгоритм умножения матриц «строка на столбец» имеет вычислительную сложность $O(N^3)$. Но главная его беда даже не в этом, а в хаотичном расположении данных. Для обеспечения максимальной локальности данных и предотвращения фрагментации памяти было принято решение полностью отказаться от стандартных двумерных массивов указателей (float).

Вместо этого мы «развернём» матрицы в плоские одномерные векторы, где вещественные и мнимые части чисел – это два элемента float, расположенных в памяти строго поочерёдно. Физический индекс элемента рассчитывается по формуле idx = i * N + j.

Генерируются матрицы псевдослучайными числами из диапазона $[0.0; 1.0)$. Чтобы заставить ядра процессора заполнять память параллельно и не мешать друг другу, мы использовали OpenMP. Сид (seed) для генератора Вихря Мерсенна (std::mt19937) внутри каждого потока – это сумма базового времени и уникального Thread ID.

Теперь то, из-за чего мне пришлось обратиться к ИИ-ассистенту, а затем во всём этом разбираться. Результатом стало то, что мы заставили процессор считать «в уме». Буквально.

Самая сложная часть – реализация непосредственно умножения матриц. Чтобы выжать максимум, мы использовали процессорные инструкции, сводящие к минимуму обращения к медленной оперативной памяти (RAM).

Мы разбили огромную матрицу на маленькие кусочки – блоки размером 64x64. Этот размер идеально подобран так, чтобы обрабатываемый блок целиком помещался в сверхбыструю кэш-память процессора (L1/L2), которая работает прямо «под рукой» у вычислительных ядер.

Далее мы задействовали широкие векторные регистры AVX2 (YMM, 256 бит) или AVX-512 (ZMM, 512 бит) для более продвинутых процессоров. В регистр YMM помещается 8 элементов float, а в ZMM – сразу 16. Но не забываем, что у нас complex float, которые лежат в памяти попарно.

Стандартный метод умножения строки на столбец тоже пришлось отмести, потому что прыгать по вертикальному столбцу для процессора – это чистое мучение, вызывающее лавину кэш-промахов.

Вместо этого мы применили схему внешнего произведения (Outer Product) с циклом вида IKJ. Алгоритм берет вертикальный микро-столбец из матрицы A высотой в 4 элемента (это называется регистровой развёрткой, или *Register Unrolling*) и «накатывает» его на горизонтальную строку матрицы B. Процессор читает данные из памяти строго последовательно, заполняя векторные регистры до упора. За один такт, используя инструкции умножения-сложения (FMA), чип считает не одну сиротливую точку, а генерирует целую вертикальную полосу результатов для матрицы C.

В результате экспериментов было выяснено, что библиотека OpenBLAS, установленная в Visual Studio через менеджер пакетов NuGet, работает примерно в 3 раза медленнее, чем OpenBLAS, устанавливаемый в систему с помощью MinGW MSYS64 (pacman).

Поэтому запуск программы в среде Visual Studio приводил к такому забавному результату:

```
[CPU] Обнаружен AVX-512. Активируем ядро ZMM.

1. Инициализация матриц данными...
-> Инициализация завершена за: 0.036186 сек.

2. Запуск cblas_cgemm (Вариант 2)...
-> BLAS завершил работу за: 3.69654 сек.

3. Запуск оптимизированного ручного алгоритма (Вариант 3)...
-> Ручной алгоритм завершил работу за: 2.85652 сек.

4. Проверка точности вычислений...
-> Норма Фробениуса разности матриц: 10.0058

================ ИТОГОВЫЕ РЕЗУЛЬТАТЫ ================
Время выполнения BLAS:        3.69654 сек.
Время выполнения ручного кода: 2.85652 сек.
Производительность BLAS:       37180.4 MFlops
Производительность ручного кода: 48114.1 MFlops
Доля ручного алгоритма от BLAS: 129.407 %

УСПЕХ! Ручной алгоритм выполнил условие (>= 30% от BLAS).
```

Да, наш ручной код обгонял промышленный BLAS.

Как только мы вынесли исполняемый файл из среды, начал использоваться другой .dll-файл, и мы получили более адекватный результат, который, тем не менее, удовлетворяет условию задачи:

```
[CPU] Обнаружен AVX-512. Активируем ядро ZMM.

1. Инициализация матриц данными...
-> Инициализация завершена за: 0.036895 сек.

2. Запуск cblas_cgemm (Вариант 2)...
-> BLAS завершил работу за: 0.798169 сек.

3. Запуск оптимизированного ручного алгоритма (Вариант 3)...
-> Ручной алгоритм завершил работу за: 2.59748 сек.

4. Проверка точности вычислений...
-> Норма Фробениуса разности матриц: 10.0041

================ ИТОГОВЫЕ РЕЗУЛЬТАТЫ ================
Время выполнения BLAS:        0.798169 сек.
Время выполнения ручного кода: 2.59748 сек.
Производительность BLAS:       172193 MFlops
Производительность ручного кода: 52912.4 MFlops
Доля ручного алгоритма от BLAS: 30.7286 %

УСПЕХ! Ручной алгоритм выполнил условие (>= 30% от BLAS).
```

ЛИСТИНГ

main:

```
#include <complex>
#include <Windows.h>
#include <cblas.h>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <omp.h>
#include <chrono>
#include <iostream>
#include <immintrin.h>
#include <algorithm>
#include <cmath>
#include <random>
#include <intrin.h>
#include "matmul.h"

using namespace std;
using namespace std::chrono;

double calculateComplexity(int n) {
    return 2.0 * pow(n, 3);
}

double calculatePerformance(double complexity, double timeInSeconds) {
    return complexity / (timeInSeconds * 1e6);
}
```

```
void multiplyMatricesBLAS(const vector<complex<float>>& A, const vector<complex<float>>&
B, vector<complex<float>>& C, const int N) {
    const auto* A_data = reinterpret_cast<const float*>(A.data());
    const auto* B_data = reinterpret_cast<const float*>(B.data());
    auto* C_data = reinterpret_cast<float*>(C.data());

    const float alpha[2] = { 1.0f, 0.0f };
    const float beta[2] = { 0.0f, 0.0f };

    cblas_cgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, N, N, N, alpha, A_data, N,
B_data, N, beta, C_data, N);
}

double calculateFrobeniusNormDiff(int N, const complex<float>* MatrixA, const
complex<float>* MatrixB) {
    double sum_sq = 0.0;
#pragma omp parallel for reduction(+:sum_sq) schedule(static)
    for (int i = 0; i < N * N; ++i) {
        float diff_real = MatrixA[i].real() - MatrixB[i].real();
        float diff_imag = MatrixA[i].imag() - MatrixB[i].imag();
        sum_sq += (diff_real * diff_real) + (diff_imag * diff_imag);
    }
    return sqrt(sum_sq);
}

// Указатель на функцию, принимающую ссылки на std::vector
typedef void (*matmul_func_ptr)(int,
    const std::vector<std::complex<float>>&,
    const std::vector<std::complex<float>>&,
    std::vector<std::complex<float>>&);

matmul_func_ptr select_best_matmul() {
    int cpuInfo[4] = { 0 };
    __cpuidex(cpuInfo, 7, 0);

    bool supports_avx512 = (cpuInfo[1] & (1 << 16)) != 0;
    bool supports_avx2 = (cpuInfo[1] & (1 << 5)) != 0;

    if (supports_avx512) {
        cout << "[CPU] Обнаружен AVX-512. Активируем ядро ZMM." << endl;
        return matmul_avx512;
    }
    if (supports_avx2) {
        cout << "[CPU] AVX-512 отсутствует. Активируем ядро AVX2 (YMM)." << endl;
        return matmul_avx2;
    }

    cout << "[CPU] Критическая ошибка: процессор не поддерживает даже AVX2!" << endl;
    return nullptr;
}

int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    // 1. Сразу при старте определяем лучшее доступное ядро
    static const matmul_func_ptr run_optimized_matmul = select_best_matmul();
    if (run_optimized_matmul == nullptr) {
        return -1;
    }

    int n = 4096;

    vector<complex<float>> A_flat(n * n);
    vector<complex<float>> B_flat(n * n);
    vector<complex<float>> C_blas(n * n);
    vector<complex<float>> C_opt(n * n);

    cout << "\n1. Инициализация матриц данными..." << endl;
```

```
    auto init_start = high_resolution_clock::now();

    unsigned int time_seed = static_cast<unsigned int>(std::time(nullptr));
#pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        std::mt19937 gen(time_seed + thread_id);
        std::uniform_real_distribution<float> dis(0.0f, 1.0f);

#pragma omp for schedule(static)
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                int idx = i * n + j;
                A_flat[idx] = complex<float>(dis(gen), dis(gen));
                B_flat[idx] = complex<float>(dis(gen), dis(gen));
            }
        }
    }
    auto init_stop = high_resolution_clock::now();
    double init_time = duration_cast<microseconds>(init_stop - init_start).count() /
1e6;
    cout << "-> Инициализация завершена за: " << init_time << " сек.\n" << endl;

    // === Запуск BLAS ===
    cout << "2. Запуск cblas_cgemm (Вариант 2)..." << endl;
    auto start = high_resolution_clock::now();
    multiplyMatricesBLAS(A_flat, B_flat, C_blas, n);
    auto stop = high_resolution_clock::now();
    double timeSecondVariant = duration_cast<microseconds>(stop - start).count() / 1e6;
    cout << "-> BLAS завершил работу за: " << timeSecondVariant << " сек.\n" << endl;

    // === Запуск Выбранного Векторного Ядра ===
    cout << "3. Запуск оптимизированного ручного алгоритма (Вариант 3)..." << endl;
    start = high_resolution_clock::now();

    // Передаем сами векторы, без .data(), так как диспетчер работает со ссылками
    run_optimized_matmul(n, A_flat, B_flat, C_opt);

    stop = high_resolution_clock::now();
    double timeThirdVariant = duration_cast<microseconds>(stop - start).count() / 1e6;
    cout << "-> Ручной алгоритм завершил работу за: " << timeThirdVariant << " сек.\n"
<< endl;

    // === Сравнение матриц ===
    cout << "4. Проверка точности вычислений..." << endl;
    double norm_diff = calculateFrobeniusNormDiff(n, C_blas.data(), C_opt.data());
    cout << "-> Норма Фробениуса разности матриц: " << norm_diff << endl;

    // Расчет метрик производительности
    double complexity = calculateComplexity(n);
    double performanceSecondVariant = calculatePerformance(complexity,
timeSecondVariant);
    double performanceThirdVariant = calculatePerformance(complexity, timeThirdVariant);

    cout << "\n================ ИТОГОВЫЕ РЕЗУЛЬТАТЫ ================" << endl;
    cout << "Время выполнения BLAS:        " << timeSecondVariant << " сек." << endl;
    cout << "Время выполнения ручного кода: " << timeThirdVariant << " сек." << endl;
    cout << "Производительность BLAS:       " << performanceSecondVariant << " MFlops"
<< endl;
    cout << "Производительность ручного кода: " << performanceThirdVariant << " MFlops"
<< endl;

    double ratio = (performanceThirdVariant / performanceSecondVariant) * 100.0;
    cout << "Доля ручного алгоритма от BLAS: " << ratio << " %" << endl;

    if (performanceThirdVariant >= 0.3 * performanceSecondVariant) {
        cout << "\nУСПЕХ! Ручной алгоритм выполнил условие (>= 30% от BLAS)." << endl;
    }
    else {
```

```
        cout << "\nТРЕБОВАНИЕ НЕ ВЫПОЛНЕНО!" << endl;
    }

    return 0;
}
```

avx2:

```
#include "matmul.h"
#include <immintrin.h>
#include <algorithm>

void matmul_avx2(int N,
    const std::vector<std::complex<float>>& A,
    const std::vector<std::complex<float>>& B,
    std::vector<std::complex<float>>& C)
{
    const int BLOCK_SIZE = 64;

    // Извлекаем сырые указатели на float из векторов
    const float* A_f = reinterpret_cast<const float*>(A.data());
    const float* B_f = reinterpret_cast<const float*>(B.data());
    float* C_f = reinterpret_cast<float*>(C.data());

    // Обнуляем матрицу C локально перед расчетами (опционально, но безопасно)
    std::fill(C_f, C_f + 2 * N * N, 0.0f);

    __m256 sign_mask = _mm256_setr_ps(-0.0f, 0.0f, -0.0f, 0.0f, -0.0f, 0.0f, -0.0f,
0.0f);

#pragma omp parallel for schedule(guided)
    for (int si = 0; si < N; si += BLOCK_SIZE) {
        for (int sk = 0; sk < N; sk += BLOCK_SIZE) {
            for (int sj = 0; sj < N; sj += BLOCK_SIZE) {

                for (int i = si; i < si + BLOCK_SIZE; i += 4) {
                    for (int k = sk; k < sk + BLOCK_SIZE; ++k) {

                        float ar0 = A_f[2 * (i * N + k)];
                        float ai0 = A_f[2 * (i * N + k) + 1];
                        __m256 vec_ar0 = _mm256_set1_ps(ar0);
                        __m256 vec_ai0 = _mm256_set1_ps(ai0);

                        float ar1 = A_f[2 * ((i + 1) * N + k)];
                        float ai1 = A_f[2 * ((i + 1) * N + k) + 1];
                        __m256 vec_ar1 = _mm256_set1_ps(ar1);
                        __m256 vec_ai1 = _mm256_set1_ps(ai1);

                        float ar2 = A_f[2 * ((i + 2) * N + k)];
                        float ai2 = A_f[2 * ((i + 2) * N + k) + 1];
                        __m256 vec_ar2 = _mm256_set1_ps(ar2);
                        __m256 vec_ai2 = _mm256_set1_ps(ai2);

                        float ar3 = A_f[2 * ((i + 3) * N + k)];
                        float ai3 = A_f[2 * ((i + 3) * N + k) + 1];
                        __m256 vec_ar3 = _mm256_set1_ps(ar3);
                        __m256 vec_ai3 = _mm256_set1_ps(ai3);

                        int j_limit = sj + BLOCK_SIZE;

                        for (int j = sj; j < j_limit; j += 4) {
                            int b_idx = 2 * (k * N + j);

                            __m256 vec_b = _mm256_loadu_ps(&B_f[b_idx]);
                            __m256 vec_b_shuf = _mm256_shuffle_ps(vec_b, vec_b,
_MM_SHUFFLE(2, 3, 0, 1));

                            // --- СТРОКА 0 ---
```

```
                                int c_idx0 = 2 * (i * N + j);
                                __m256 vec_c0 = _mm256_loadu_ps(&C_f[c_idx0]);
                                vec_c0 = _mm256_fmadd_ps(vec_ar0, vec_b, vec_c0);
                                __m256 i_term0 = _mm256_mul_ps(vec_ai0, vec_b_shuf);
                                i_term0 = _mm256_xor_ps(i_term0, sign_mask);
                                vec_c0 = _mm256_add_ps(vec_c0, i_term0);
                                _mm256_storeu_ps(&C_f[c_idx0], vec_c0);

                                // --- СТРОКА 1 ---
                                int c_idx1 = 2 * ((i + 1) * N + j);
                                __m256 vec_c1 = _mm256_loadu_ps(&C_f[c_idx1]);
                                vec_c1 = _mm256_fmadd_ps(vec_ar1, vec_b, vec_c1);
                                __m256 i_term1 = _mm256_mul_ps(vec_ai1, vec_b_shuf);
                                i_term1 = _mm256_xor_ps(i_term1, sign_mask);
                                vec_c1 = _mm256_add_ps(vec_c1, i_term1);
                                _mm256_storeu_ps(&C_f[c_idx1], vec_c1);

                                // --- СТРОКА 2 ---
                                int c_idx2 = 2 * ((i + 2) * N + j);
                                __m256 vec_c2 = _mm256_loadu_ps(&C_f[c_idx2]);
                                vec_c2 = _mm256_fmadd_ps(vec_ar2, vec_b, vec_c2);
                                __m256 i_term2 = _mm256_mul_ps(vec_ai2, vec_b_shuf);
                                i_term2 = _mm256_xor_ps(i_term2, sign_mask);
                                vec_c2 = _mm256_add_ps(vec_c2, i_term2);
                                _mm256_storeu_ps(&C_f[c_idx2], vec_c2);

                                // --- СТРОКА 3 ---
                                int c_idx3 = 2 * ((i + 3) * N + j);
                                __m256 vec_c3 = _mm256_loadu_ps(&C_f[c_idx3]);
                                vec_c3 = _mm256_fmadd_ps(vec_ar3, vec_b, vec_c3);
                                __m256 i_term3 = _mm256_mul_ps(vec_ai3, vec_b_shuf);
                                i_term3 = _mm256_xor_ps(i_term3, sign_mask);
                                vec_c3 = _mm256_add_ps(vec_c3, i_term3);
                                _mm256_storeu_ps(&C_f[c_idx3], vec_c3);
                            }
                        }
                    }
                }
            }
        }
}
```

avx512:

```
#include "matmul.h"
#include <immintrin.h>
#include <algorithm>

void matmul_avx512(int N,
    const std::vector<std::complex<float>>& A,
    const std::vector<std::complex<float>>& B,
    std::vector<std::complex<float>>& C)
{
    const int n = N;
    const int BLOCK_SIZE = 64;

    // Переходим на работу с сырыми структурами внутри вектора
    const std::complex<float>* A_flat = A.data();
    const std::complex<float>* B_flat = B.data();
    std::complex<float>* C_flat = C.data();

    // Быстро обнуляем результирующий массив перед накоплением сумм
    std::fill(C_flat, C_flat + n * n, std::complex<float>(0.0f, 0.0f));

    // Маска знаков для ZMM
    __m512 sign_mask = _mm512_set_ps(
        0.0f, -0.0f, 0.0f, -0.0f, 0.0f, -0.0f, 0.0f, -0.0f,
        0.0f, -0.0f, 0.0f, -0.0f, 0.0f, -0.0f, 0.0f, -0.0f
```

```
    );

#pragma omp parallel for schedule(static)
    for (int si = 0; si < n; si += BLOCK_SIZE) {
        for (int sk = 0; sk < n; sk += BLOCK_SIZE) {
            for (int sj = 0; sj < n; sj += BLOCK_SIZE) {

                for (int i = si; i < si + BLOCK_SIZE; i += 4) {
                    for (int k = sk; k < sk + BLOCK_SIZE; ++k) {

                        __m512 vec_ar0 = _mm512_set1_ps(A_flat[i * n + k].real());
                        __m512 vec_ai0 = _mm512_set1_ps(A_flat[i * n + k].imag());

                        __m512 vec_ar1 = _mm512_set1_ps(A_flat[(i + 1) * n + k].real());
                        __m512 vec_ai1 = _mm512_set1_ps(A_flat[(i + 1) * n + k].imag());

                        __m512 vec_ar2 = _mm512_set1_ps(A_flat[(i + 2) * n + k].real());
                        __m512 vec_ai2 = _mm512_set1_ps(A_flat[(i + 2) * n + k].imag());

                        __m512 vec_ar3 = _mm512_set1_ps(A_flat[(i + 3) * n + k].real());
                        __m512 vec_ai3 = _mm512_set1_ps(A_flat[(i + 3) * n + k].imag());

                        for (int j = sj; j < sj + BLOCK_SIZE; j += 8) {
                            int idx_b = k * n + j;

                            __m512 vec_b = _mm512_loadu_ps(reinterpret_cast<const
float*>(&B_flat[idx_b]));
                            __m512 vec_b_shuf = _mm512_shuffle_ps(vec_b, vec_b,
_MM_SHUFFLE(2, 3, 0, 1));

                            // --- Строка C0 ---
                            int idx_c0 = i * n + j;
                            __m512 vec_c0 =
_mm512_loadu_ps(reinterpret_cast<float*>(&C_flat[idx_c0]));
                            vec_c0 = _mm512_fmadd_ps(vec_ar0, vec_b, vec_c0);
                            __m512 i_term0 = _mm512_mul_ps(vec_ai0, vec_b_shuf);
                            i_term0 = _mm512_xor_ps(i_term0, sign_mask);
                            vec_c0 = _mm512_add_ps(vec_c0, i_term0);
                            _mm512_storeu_ps(reinterpret_cast<float*>(&C_flat[idx_c0]),
vec_c0);

                            // --- Строка C1 ---
                            int idx_c1 = (i + 1) * n + j;
                            __m512 vec_c1 =
_mm512_loadu_ps(reinterpret_cast<float*>(&C_flat[idx_c1]));
                            vec_c1 = _mm512_fmadd_ps(vec_ar1, vec_b, vec_c1);
                            __m512 i_term1 = _mm512_mul_ps(vec_ai1, vec_b_shuf);
                            i_term1 = _mm512_xor_ps(i_term1, sign_mask);
                            vec_c1 = _mm512_add_ps(vec_c1, i_term1);
                            _mm512_storeu_ps(reinterpret_cast<float*>(&C_flat[idx_c1]),
vec_c1);

                            // --- Строка C2 ---
                            int idx_c2 = (i + 2) * n + j; // Строка очищена от мусора
                            __m512 vec_c2 =
_mm512_loadu_ps(reinterpret_cast<float*>(&C_flat[idx_c2]));
                            vec_c2 = _mm512_fmadd_ps(vec_ar2, vec_b, vec_c2);
                            __m512 i_term2 = _mm512_mul_ps(vec_ai2, vec_b_shuf);
                            i_term2 = _mm512_xor_ps(i_term2, sign_mask);
                            vec_c2 = _mm512_add_ps(vec_c2, i_term2);
                            _mm512_storeu_ps(reinterpret_cast<float*>(&C_flat[idx_c2]),
vec_c2);

                            // --- Строка C3 ---
                            int idx_c3 = (i + 3) * n + j;
                            __m512 vec_c3 =
_mm512_loadu_ps(reinterpret_cast<float*>(&C_flat[idx_c3]));
                            vec_c3 = _mm512_fmadd_ps(vec_ar3, vec_b, vec_c3);
                            __m512 i_term3 = _mm512_mul_ps(vec_ai3, vec_b_shuf);
```

```
                            i_term3 = _mm512_xor_ps(i_term3, sign_mask);
                            vec_c3 = _mm512_add_ps(vec_c3, i_term3);
                            _mm512_storeu_ps(reinterpret_cast<float*>(&C_flat[idx_c3]),
vec_c3);
                        }
                    }
                }
            }
        }
    }
}
```

matmul.h

```
#pragma once
#include <vector>
#include <complex>

// Прототип функции для AVX2
void matmul_avx2(int n,
    const std::vector<std::complex<float>>& A,
    const std::vector<std::complex<float>>& B,
    std::vector<std::complex<float>>& C);

// Прототип функции для AVX-512
void matmul_avx512(int n,
    const std::vector<std::complex<float>>& A,
    const std::vector<std::complex<float>>& B,
    std::vector<std::complex<float>>& C);
```

ВЫВОД

В ходе выполнения лабораторной работы были детально изучены и применены на практике современные методы низкоуровневой оптимизации работы с большими данными.

Полученный результат (как работы программы, так и моей работы) очень неплох. Хотя я и использовал ИИ-ассистента, было потрачено достаточно много времени, чтобы разобраться в принципах работы алгоритма.