Национальный исследовательский ядерный университет «МИФИ» (Московский Инженерно–Физический Институт)
Кафедра №42 «Криптология и кибербезопасность»

Лабораторная работа №1:
«**Цена вызова**»

Антон Гатченко Б22-525
2025 г.

Используемая рабочая среда:

- Процессор - AMD Ryzen 5 5600H (laptop), 6c/12t
- Оперативная память – DDR4 16 ГБ
- ОС - Windows 10 Pro 22H2 19045.4780, 64 bit
- IDE – GCC/G++ 13.1, OpenMP 201511

Ход работы:

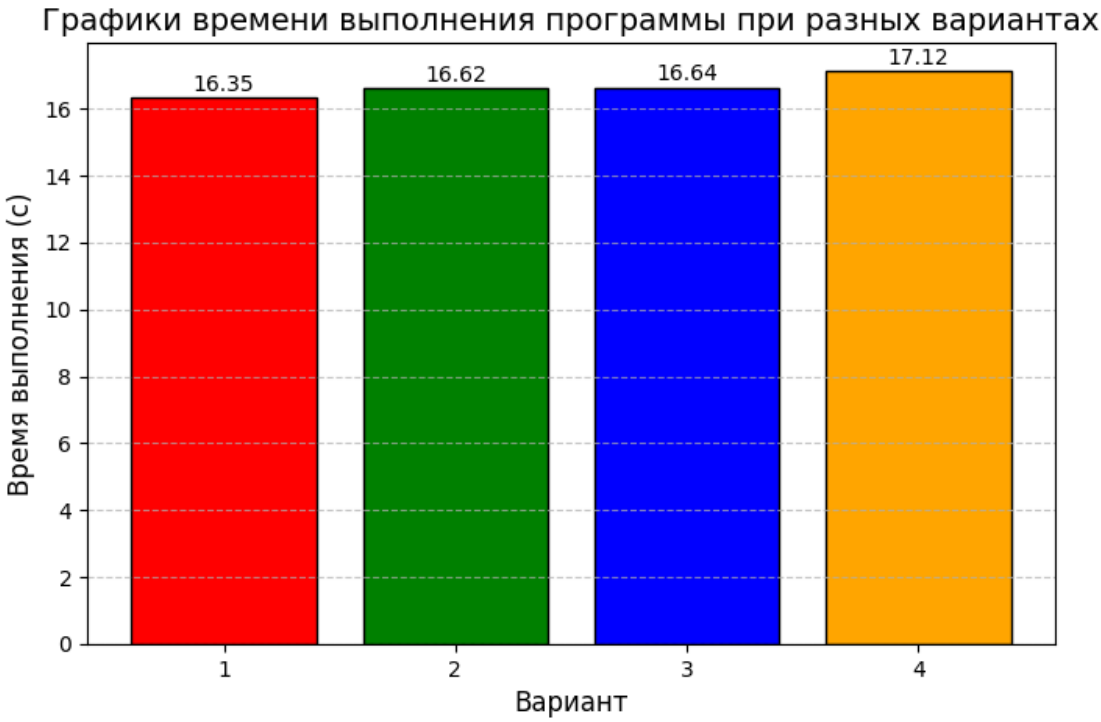
Для написания программы использовался язык C++.

Замеры времени выполнения всех вариантов программы проводились на $3 * 10^8$ уравнениях, большее количество не удалось поставить из-за недостатка оперативной памяти. Каждый коэффициент уравнения был задан типом float, значение варьировалось от -10^6 до 10^6. Коэффициенты уравнения генерировались случайно, но итоговый набор был одинаковым для всех вариантов программы.

При компиляции программы флаги оптимизации *не* выставлялись.

Для 1 и 2 варианта код реализовывался в одном файле main.cpp; для 3 и 4 – в трёх файлах, main.cpp, equation_solver.cpp, equation_solver.h.

График времени выполнения программы при разных вариантах:



(1 – обычный, 2 – inline, 3 – две разных компиляции, 4 – передача обоих названий компилятору)

Заключение:

В ходе данной лабораторной работы было протестировано влияние подхода к компоновке на время работы программы.

Все варианты показали довольно схожие результаты, максимальное отличие оказалось между 1 (все функции в одном файле, без inline) и 4 (функции в разных файлах, компилятору переданы сразу все имена файлов) вариантами – 0.77 секунды или 4.7%, 16.35 с. в 1 варианте и 17.12 с. во втором

Варианты 2 (все функции в одном файле, с inline) и 3 (скомпилированы оба файла и из них скомпонована программа) показали практически идентичные результаты, 16.62 и 16.64 секунды соответственно.

Малое отличие между вариантами может быть вызвано компилятором, который может вносить свои изменения и оптимизации. Лучшая производительность достигается, когда весь код находится в одном файле, так как компилятор может применять глобальные оптимизации, однако современные компиляторы хорошо оптимизируют даже код в разных файлах, снижая влияние подхода к компоновке на время работы программы.

Приложение:

1. Исходный код программы с измерением времени работы программы:

```
#include <iostream>
#include <cmath>
#include <vector>
#include <random>
#include <chrono>
#include <immintrin.h>
#include <omp.h>

#define SIZE 3e8
#define SEED 15032025

using std::vector, std::array, std::cout, std::endl;

std::vector<std::array<double, 3>> generateRandomVectorsAVX(int count, int
seed) {
    std::cout << "Generating " << count << " random vectors" << std::endl;
    const auto start = std::chrono::high_resolution_clock::now();

    alignas(32) std::vector<std::array<double, 3>> result(count);
    __m256d scale = _mm256_set1_pd(1e6);

#pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        std::mt19937 generator(seed + thread_id);
        std::uniform_real_distribution<double> distribution(-1e6, 1e6);

#pragma omp for
        for (int i = 0; i < count; ++i) {
            __m256d rnd = _mm256_set_pd(
                distribution(generator),
                distribution(generator),
                distribution(generator),
                0.0 // Заполнитель для выравнивания
            );
            rnd = _mm256_mul_pd(rnd, scale);
            _mm256_store_pd(result[i].data(), rnd);
        }
    }

    const auto end = std::chrono::high_resolution_clock::now();
    const std::chrono::duration<double> elapsed = end - start;
    std::cout << "Finished generating after " << elapsed.count() << "
seconds" << std::endl;

    return result;
}

double calculateDiscriminant(double a, double b, double c){
    return b * b - 4 * a * c;
}
```

```
double calculateRoot1(double a, double b, double discriminant){
    if (discriminant >= 0){
        return (-b + sqrt(discriminant)) / (2 * a);
    }
    return nan("");
}

double calculateRoot2(double a, double b, double discriminant){
    if (discriminant >= 0){
        return (-b - sqrt(discriminant)) / (2 * a);
    }
    return nan("");
}

void timeCode(){
    vector<array<double, 3>> equations = generateRandomVectorsAVX(SIZE,
SEED);
    const auto start = std::chrono::high_resolution_clock::now();

    for (const auto &eq: equations){
        const double a = eq[0];
        const double b = eq[1];
        const double c = eq[2];

        const double discriminant = calculateDiscriminant(a, b, c);
        const double root1 = calculateRoot1(a, b, discriminant);
        const double root2 = calculateRoot2(a, b, discriminant);

        // cout << a << "x^2 + " << b << "x + " << c << " = 0" << endl;
        // if (discriminant < 0){
        //     cout << "D < 0" << endl;
        // } else{
        //     cout << "Roots: x1 = " << root1 << ", x2 = " << root2 <<
endl;
        // }
        // cout << endl;
    }

    const auto end = std::chrono::high_resolution_clock::now();
    const std::chrono::duration<double> elapsed = end - start;
    cout << "Elapsed time: " << elapsed.count() << " seconds" << endl;
}

int main(){
    timeCode();
    return 0;
}
```

2. CMakeLists.txt для 3го варианта:

```
cmake_minimum_required(VERSION 3.29)

project(EquationSolver)

set(CMAKE_CXX_STANDARD 20)
```

```
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Добавление флага для AVX
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -mavx")

find_package(OpenMP REQUIRED)
if(OPENMP_FOUND)
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fopenmp")
endif()

# Создание статической библиотеки из equation_solver.cpp
add_library(equation_solver STATIC equation_solver.cpp)

# Создание исполняемого файла main
add_executable(EquationSolver main.cpp)

# Связывание библиотеки equation_solver с исполняемым файлом
target_link_libraries(EquationSolver PRIVATE equation_solver)

# Указание заголовочных файлов
target_include_directories(EquationSolver PUBLIC
${CMAKE_CURRENT_SOURCE_DIR})
```

3. CMakeLists.txt для 4го варианта:

```
cmake_minimum_required(VERSION 3.29)

project(EquationSolver)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Добавление флага для AVX
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -mavx")

# Поиск OpenMP
find_package(OpenMP REQUIRED)
if(OPENMP_FOUND)
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fopenmp")
endif()

# Добавление исполняемого файла
add_executable(EquationSolver main.cpp equation_solver.cpp)

# Указание заголовочных файлов
target_include_directories(EquationSolver PUBLIC
${CMAKE_CURRENT_SOURCE_DIR})
```