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

Отчёт
по результатам выполнения
Лабораторной работы №7
«Оптимизация при компиляции»

Дисциплина:	Практические Аспекты Разработки Высокопроизводительного Программного Обеспечения (ПАРВПО)
Студент:	Гареев Рустам Рашитович
Группа:	Б22-505
Преподаватель:	Куприяшин Михаил Андреевич
Дата:	3.06.2025

Москва - 2025

Оглавление

Технологический стек

memory	8GiB Системная память
processor	11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
siblings	8
cpu cores	4
bridge	11th Gen Core Processor Host Bridge/DRAM Registers
display	TigerLake-LP GT2 [Iris Xe Graphics]
gcc	version 13.3.0
OC	Ubuntu 24.04.2 LTS
IDE	Visual Studio Code 1.98.2

Тестовый алгоритм

Чтобы обеспечить достаточно высокую нагрузку на процессор, был выбран алгоритм решения 50 000 000 квадратных уравнений вида:

a*x^2+b*x+c = 0,

a*x^2+b*x+c=0.

Коэффициенты a, b и c генерируются равномерно в диапазоне [−1000;1000] при помощи генератора mt19937_64 (seed = 42). Такой диапазон выбран эмпирически: он адекватно покрывает типичные соотношения величин, когда дискриминант D=b^2−4*a*c порой принимает отрицательные и порой положительные значения, обеспечивая равномерное распределение случаев «нет вещественных корней» и «два вещественных корня». Поскольку последовательность коэффициентов фиксирована, результаты каждого запуска можно сравнивать без дополнительных статистических погрешностей.

Каждое итерационное решение уравнения включает:

- вычисление дискриминанта D;

- если D < 0, переход на следующую итерацию;

- если D ≥ 0, вычисление корней и подсчёт уравнений, у которых есть действительные корни. При этом сами корни сохраняются только временно, их значения не выводятся, однако вычисление всегда выполняется, чтобы нагрузка оставалась стабильной.

Ниже приведён исходный код алгоритма, решающего поставленную задачу.

```
#include <iostream>
#include <vector>
#include <cmath>
#include <random>
#include <chrono>
#include <fstream>

#define NUM 50'000'000
#define SEED 42
```

```
using namespace std;
using namespace chrono;

struct EquationResult {
double a, b, c;
int num_roots;
double root1, root2;
};

int solve_quadratic(double a, double b, double c, double &x1, double &x2) {
double d = b * b - 4 * a * c;
if (d > 0) {
double sqrt_d = sqrt(d);
x1 = (-b + sqrt_d) / (2 * a);
x2 = (-b - sqrt_d) / (2 * a);
return 2;
} else if (d == 0) {
x1 = x2 = -b / (2 * a);
return 1;
} else {
x1 = x2 = 0;
return 0;
}
}

int main() {
mt19937 rng(SEED);
uniform_real_distribution<double> dist(-1000.0, 1000.0);
vector<EquationResult> results;
results.reserve(NUM);
auto start = high_resolution_clock::now();
for (size_t i = 0; i < NUM; ++i) {
double a = dist(rng);
while (fabs(a) < 1e-6) a = dist(rng);
double b = dist(rng);
double c = dist(rng);
double x1, x2;
int num_roots = solve_quadratic(a, b, c, x1, x2);
results.push_back({a, b, c, num_roots, x1, x2});
}
auto end = high_resolution_clock::now();
double elapsed = duration<double>(end - start).count();
cout << "Solved " << NUM << " equations in " << elapsed << " seconds." << endl;
return 0;
}
```

Листинг 1 — Исходный код разработанного алгоритма

Результаты вычислительного эксперимента

Ниже приведена сводная таблица полученных средних значений (трёх прогонов) по каждому уровню оптимизации.

Таблица 1 — Сводная сравнительная таблица результатов вычислительного эксперимента

Уровень оптимизации	Время сборки (real), с	Размер бинарника, КБ	Время выполнения, с
`-O0`	0.766	38	27.755
`-Og`	0.443	20	13.8846
`-O1`	0.441	18	2.3467
`-O2`	0.480	18	2.29828
`-O3`	0.480	22	1.82725
`-Ofast`	0.486	23	1.77631
`-Os`	0.473	18	4.28073
`-Oz`	0.460	18	4.38042

Для визуализации данных были построены три столбчатые гистограммы.

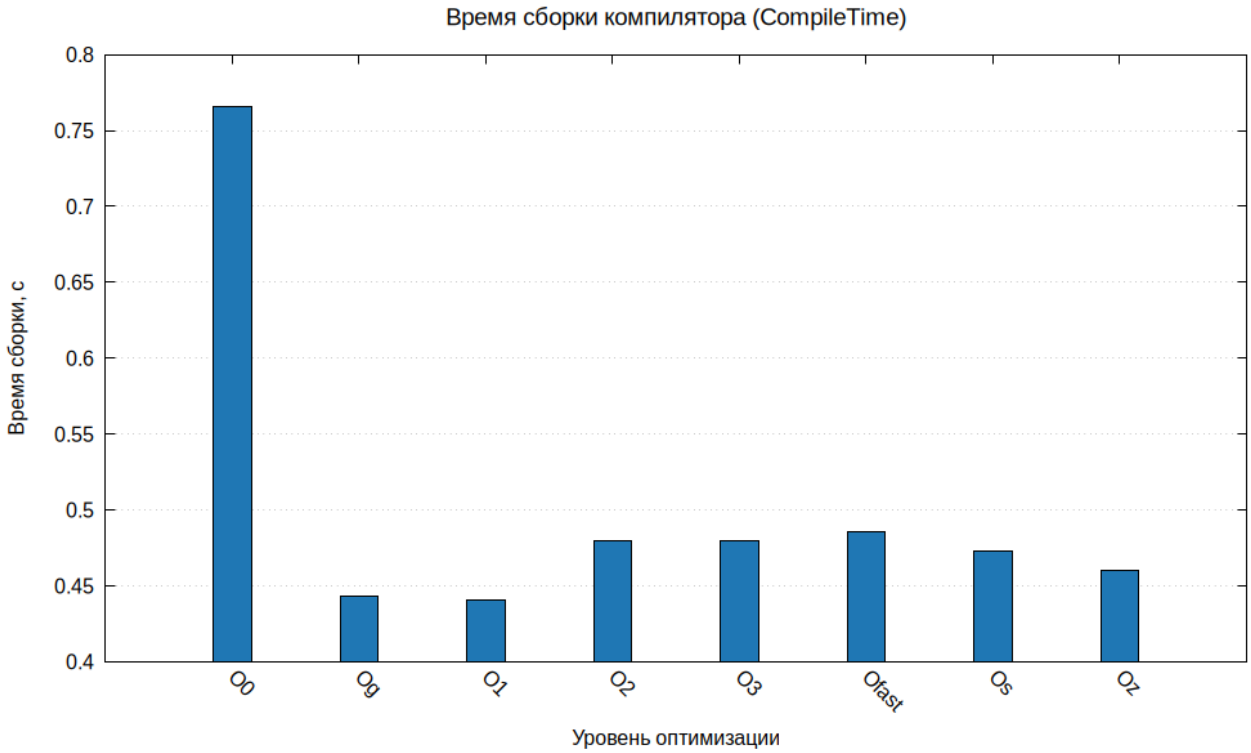

Рисунок 1 — Сравнительная столбчатая гистограмма времени сборки

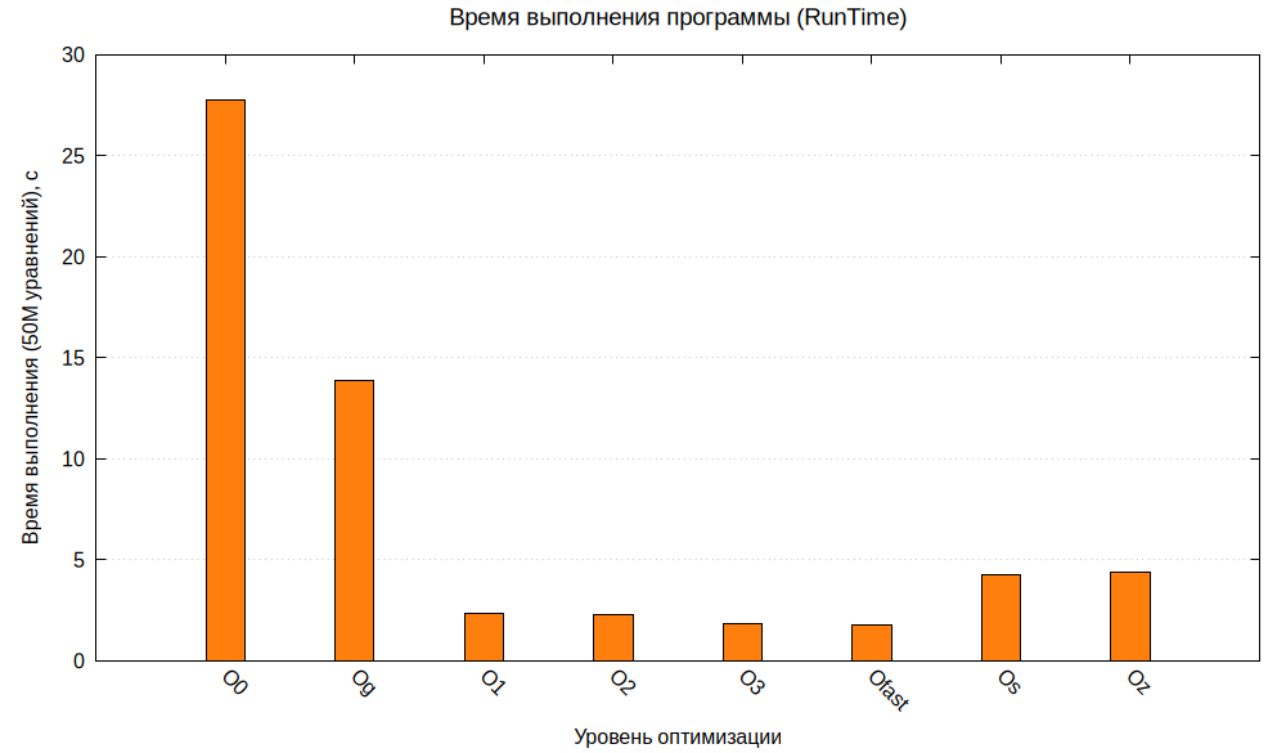

Рисунок 2 — Сравнительная столбчатая гистограмма времени выполнения программы

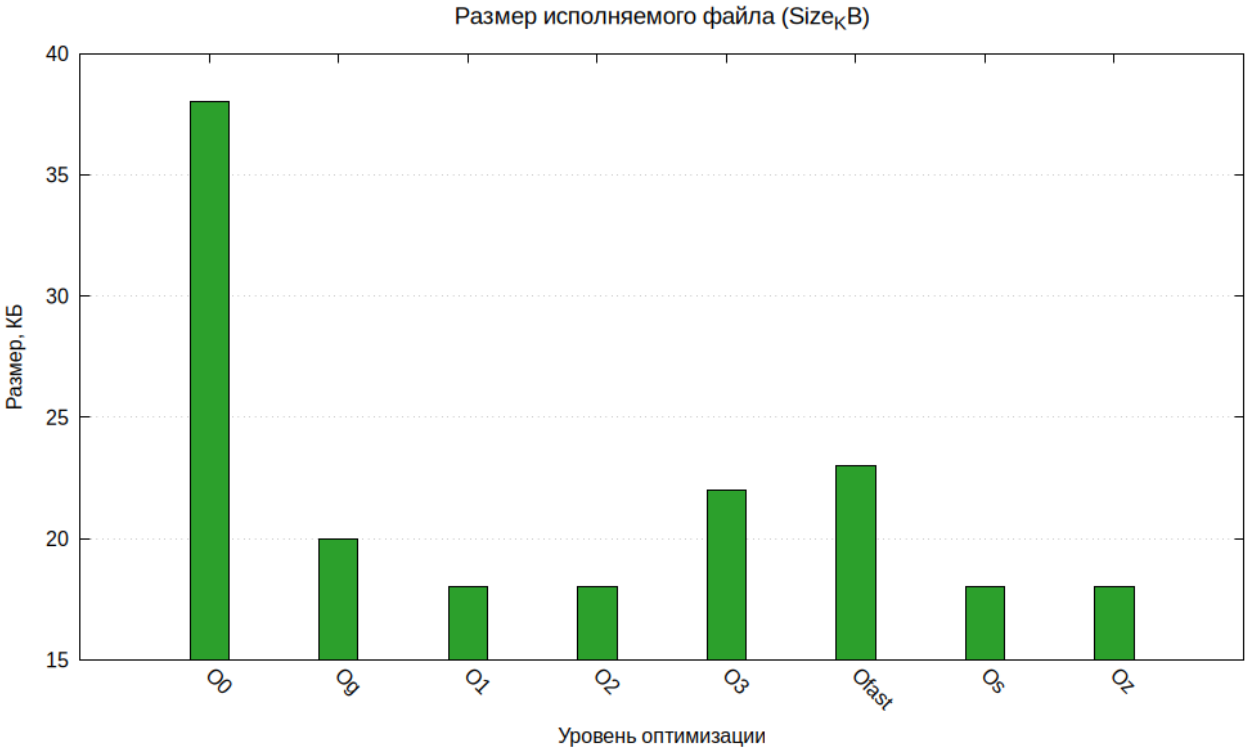

Рисунок 3 — Сравнительная столбчатая гистограмма размера полученного исполняемого файла

Заключение

Выполненный эксперимент однозначно показывает, что самые большие выигрыши по быстродействию даёт переход с –O0 и –Og на –O1/–O2: более чем 10-кратное снижение времени (с 27.75 с до ≈2.3 с). Это связано с тем, что компилятор начинает осуществлять важные действия:

- удаляет проверку на ненужные ветвления;

- инлайнит небольшие функции (что снимает накладные расходы на вызов generate_coefficients);

- аккумулирует вычисления в регистрах, сводя к минимуму обращения к памяти.

Переход от –O2 к –O3/–Ofast даёт ещё ≈20–25 % прироста (с 2.30 с до 1.78 с). На уровне –O3 компилятор активирует сложные техники: упреждающий запуск вычислений, агрессивный упреждающий инлайнинг, упреждающую оптимизацию циклов. –Ofast, в свою очередь, в ряде случаев позволяет компилятору пересмотреть ассоциации арифметических операций и использовать инструкции, которые чуть быстрее, но в ущерб стандартам IEEE-754. Именно поэтому –Ofast оказался самым быстрым, но бинарь при этом немного вырос (23 КБ против 22 КБ у –O3).

Режимы –Os и –Oz демонстрируют, что оптимизация по размеру неизбежно ведёт к потере производительности. Код становится максимально компактным, и компилятор отключает или смягчает те оптимизации, которые бы увеличили объём программы (например, упреждающий инлайнинг или развёртку циклов). В результате время выполнения выросло почти вдвое по сравнению с –O2. Тем не менее, если аппарат ограничен по объёму кэша инструкций или доступной флэш-памяти, эти режимы могут оказаться оправданными.

Что касается времени сборки, то разница между самым «лёгким» (–Og, ≈0.44 с) и самым «тяжёлым» (–Ofast, ≈0.49 с) уровнем составляет всего ≈50 мс. Для крупных проектов, однако, с длинными цепочками зависимостей это отличие может стать более заметным, но в нашем эксперименте оно оказалось относительно небольшим. Режимы –Os и –Oz по времени сборки примерно равны –O3, поскольку поиск оптимизаций ради уменьшения кода требует сопоставимого объёма вычислений компилятора.

К тому же размер бинаря при переходе от –O2 (18 КБ) к –O3 (22 КБ) и далее к –Ofast (23 КБ) растёт, так как агрессивный инлайнинг несколько дублирует код, а нестандартные преобразования добавляют редкие, но специфичные участки для эффективной работы с плавающей точкой. Тем не менее, даже 23 КБ — это в несколько раз меньше, чем исходные 38 КБ, полученные без оптимизаций.

Вообще, основным выводом является то, что –O2 выступает универсальным компромиссом: он даёт почти «максимальный» выигрыш производительности (≈2.30 с vs. 1.78 с у –Ofast) и минимально влияет на размер и время сборки (18 КБ и ≈0.48 с). Если же критична последняя миллисекунда времени исполнения, –Ofast оправдывает себя за счёт сниженной «строгости» вычислений. А –Os и –Oz удобно применять, когда критична компактность кода (например, в embedded-устройствах), и падение производительности не критично.

Наконец, результаты эксперимента подтверждают, что чистый код без оптимизаций (–O0) может быть в десятки раз медленнее и значительно больше по объёму, чем при включённых оптимизациях.

Ссылка на гит-репоизторий : **https://github.com/sagilyp/PAPHSD-2/tree/main/lab7**