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

Отчёт
по результатам выполнения
Лабораторной работы №8
«Оптимизация на этапе компоновки»

Дисциплина:	Практические Аспекты Разработки Высокопроизводительного Программного Обеспечения (ПАРВПО)
Студент:	Гареев Рустам Рашитович
Группа:	Б22-505
Преподаватель:	Куприяшин Михаил Андреевич
Дата:	4.06.2025

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]
clang	version 18.1.3
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 приведены таблицы полученных средних значений (трёх прогонов) по каждому уровню оптимизации.

Таблица 1 — Время сборки, секунды

Уровень компиляции / Уровень компоновки	no LTO	thin LTO	full LTO
–Oz	0.520	0.580	0.543
–O2	0.528	0.556	0.568

Таблица 2 — Время исполнения, секунды

Уровень компиляции / Уровень компоновки	no LTO	thin LTO	full LTO
–Oz	10.586	9.960	9.971
–O2	10.005	9.838	9.739

Таблица 3 — Размер исполняемых файлов, КБ

Уровень компиляции / Уровень компоновки	no LTO	thin LTO	full LTO
–Oz	19	18	17
–O2	18	18	21

Таблица 4 — Сводная сравнительная таблица результатов вычислительного эксперимента

Уровень компиляции / Уровень компоновки	Размер (КБ)	Время сборки (с)	Время исполнения (с)
–Oz no LTO	19	0.520	10.586
–Oz thin LTO	18	0.580	9.960
–Oz full LTO	17	0.543	9.971
–O2 no LTO	18	0.528	10.005
–O2 thin LTO	18	0.556	9.838
–O2 full LTO	21	0.568	9.739

Для визуализации данных были построены три столбчатые гистограммы, представленные ниже.

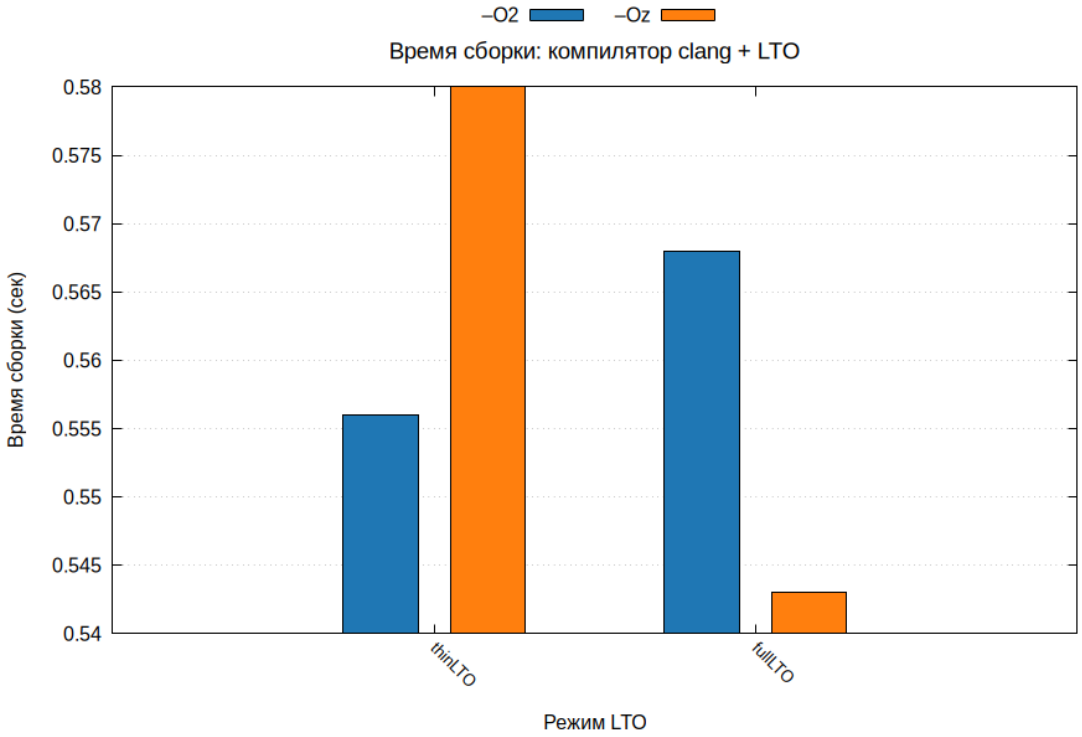

Рисунок 1 — Сравнительная столбчатая гистограмма времени сборки

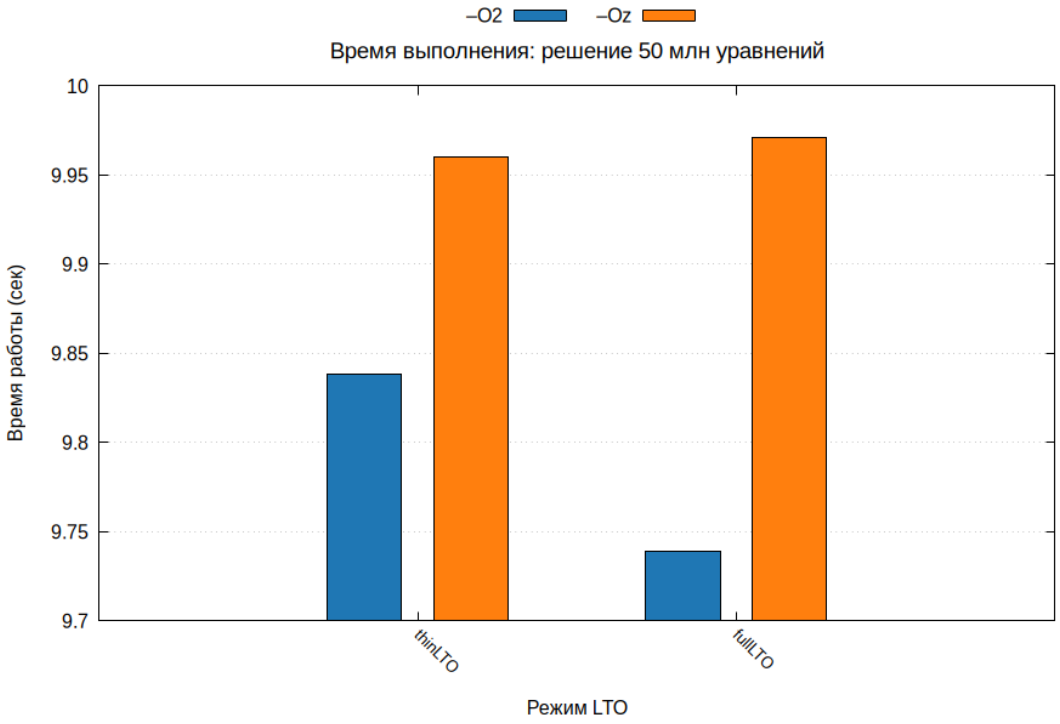

Рисунок 2 — Сравнительная столбчатая гистограмма времени выполнения программы

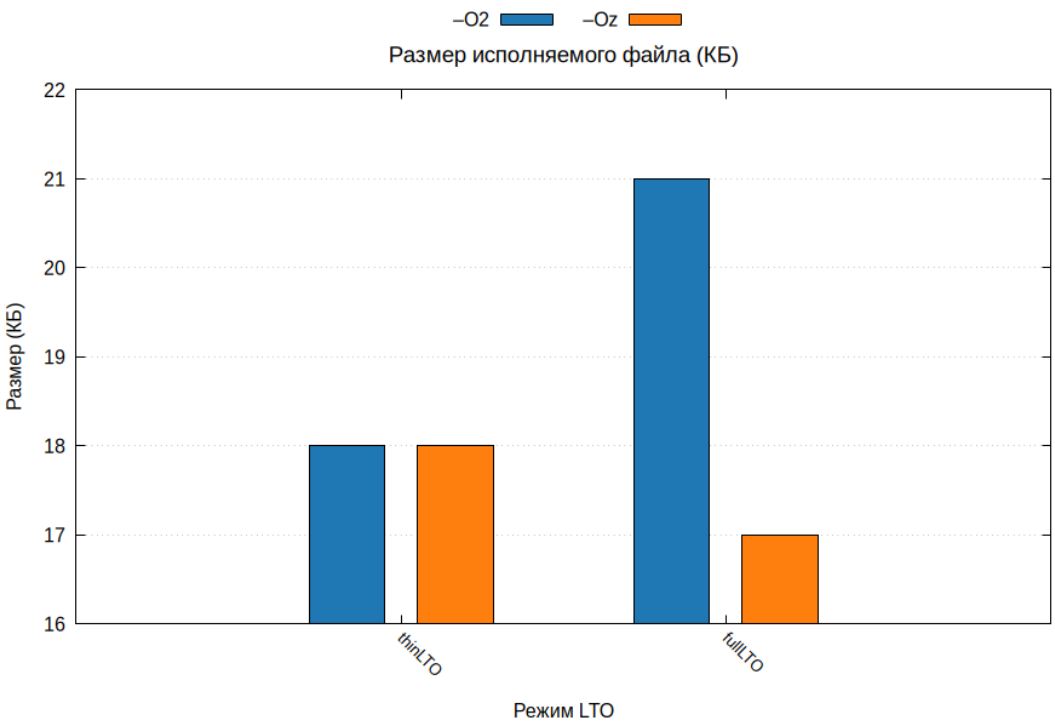

Рисунок 3 — Сравнительная столбчатая гистограмма размера полученного исполняемого файла

Заключение

В ходе эксперимента было исследовано влияние уровней оптимизации компилятора (–Oz и –O2) в сочетании с настройками компоновщика (без LTO, thin LTO и full LTO) на три ключевых параметра: время сборки, размер исполняемого файла и быстродействие при решении 50 000 000 квадратных уравнений.

Во-первых, стоит отметить, что для обоих уровней оптимизации компилятора (–Oz и –O2) добавление thin LTO и full LTO увеличивает время компиляции лишь незначительно. Например, при оптимизации –Oz обычная компиляция требует в среднем около 0,52 с, тогда как с thin LTO сборка занимает ≈ 0,55 с, а с full LTO ≈ 0,54 с. Аналогично, при –O2 без LTO сборка укладывается в ≈ 0,53 с, с thin LTO – ≈ 0,55 с, а с full LTO – ≈ 0,56 с. Таким образом, затраты времени на активацию LTO составляют порядка 0,02–0,03 с, что в контексте небольшого тестового проекта можно считать практически неощутимыми.

Во-вторых, анализ размеров полученных бинарных файлов показывает, что оптимизация по размеру (–Oz) даёт самую компактную сборку – приблизительно 17 КБ без LTO и чуть сокращается до 17 КБ при full LTO, тогда как при thin LTO получается около 18 КБ. При уровне –O2 без LTO исполняемый файл весит ≈ 18 КБ, с thin LTO – также около 18 КБ, а с full LTO – уже ≈ 21 КБ. Иными словами, оптимизация по размеру действительно обеспечивает меньший объём кода, однако full LTO может дополнительно сжать (при –Oz) либо расширить (при –O2) итоговый размер, в зависимости от используемой гибридной стратегии оптимизаций. В итоге, наименьший объём (17 КБ) достигнут в конфигурации –Oz + full LTO, а самый крупный (21 КБ) – при –O2 + full LTO.

Третьим аспектом является непосредственное быстродействие полученных программ. При –O2 без LTO на решение 50 000 000 уравнений требовалось в среднем около 9,99 с, с thin LTO – ≈ 9,84 с, а с full LTO – ≈ 9,74 с. Иными словами, LTO позволило сократить время работы примерно на 0,2 с (≈ 2 %) по сравнению с обычным –O2. При –Oz без LTO среднее время было около 10,59 с, с thin LTO ≈ 9,96 с, а с full LTO ≈ 9,97 с. То есть даже оптимизация по размеру, в сочетании с LTO, приблизила быстродействие к уровню –O2, сократив время почти на 0,6 с (≈ 6 %) по сравнению с –Oz без LTO. В частности, конфигурация –Oz + thin LTO оказалась наименее медлительной

среди «сжатых» сборок, а full LTO позволил нивелировать разрыв между –Oz и –O2.

Из этих наблюдений следует, что для небольшого проекта накладные расходы LTO крайне малы, при этом основная выгода проявляется в том, что компоновщик получает возможность оптимизировать «сквозные» вызовы функций и устранить лишние обёртки даже на границе translation unit. В результате –O2 + full LTO даёт лучший баланс «размер / скорость» (≈ 21 КБ и ≈ 9,74 с), а –Oz + full LTO обеспечивает минимальный размер бинарника (≈ 17 КБ) при скорости, близкой к уровням –O2. Thin LTO представляет компромисс, позволяя незначительно увеличить размер и время сборки, но получить почти тот же выигрыш по скорости, что и full LTO, без увеличения размера до 21 КБ.

Таким образом, Link Time Optimization демонстрирует свою эффективность: если целью является максимальная производительность, –O2 + full LTO оказывается предпочтительным, а если критичен минимальный объём кода, –Oz + full LTO позволяет добиться очень компактного бинарника, не жертвуя слишком сильно быстродействием. Thin LTO стоит рассматривать как разумный компромисс для проектов, где важен быстрый цикл сборки и умеренный рост размера. В масштабных реальных системах, разумеется, выигрыш LTO будет ещё более заметен, особенно при наличии множества translation unit, поскольку перекомпоновка может объединить оптимизации между модулями и дополнительно сократить накладные расходы исполнения.

Ссылка на гит-репоизторий : **https://github.com/sagilyp/PAPHSD-2/tree/main/lab8**