Министерство науки и высшего образования Российской Федерации

Новосибирский государственный технический университет

Кафедра параллельных вычислительных технологий

Расчетно-графическое задание

По дисциплине

«Архитектура вычислительных систем и системное программирование»

Факультет: ПМИ

Группа: ПМИ-11

Студент: Барковский И.Н.

Преподаватель: Перепелкин В.А.

Новосибирск

2022

**Цель работы:** исследовать зависимость времени работы прикладных программ от уровня оптимизации компилятора.

**Код:**

#include <clocale>

#include <random>

#include <chrono>

#include <vector>

#include <iostream>

std::vector<float> multiplying(int n) {

srand(time(NULL));

std::vector<float> first\_matrix(n \* n);

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

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

first\_matrix[i \* n + j] = std::rand() % 100;

}

}

std::vector<float> second\_matrix(n \* n);

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

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

second\_matrix[i \* n + j] = std::rand() % 100;;

}

}

std::vector<float> result\_matrix(n \* n);

auto t1 = std::chrono::high\_resolution\_clock::now();

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

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

result\_matrix[i \* n + j] = 0;

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

result\_matrix[i \* n + j] += first\_matrix[i \* n + k] \* second\_matrix[k \* n + j];

}

}

}

auto t2 = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> time = (t2 - t1);

std::cout << "Для матриц размером " << n << ": " << time.count() << " секунд" << std::endl;

return result\_matrix;

}

std::vector<float> transposed\_multiplying(int n) {

srand(time(NULL));

std::vector<float> first\_matrix(n \* n);

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

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

first\_matrix[i \* n + j] = std::rand() % 100;

}

}

std::vector<float> second\_matrix(n \* n);

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

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

second\_matrix[i \* n + j] = std::rand() % 100;;

}

}

std::vector<float> result\_matrix(n \* n);

auto t1 = std::chrono::high\_resolution\_clock::now();

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

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

result\_matrix[i \* n + j] = 0;

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

result\_matrix[i \* n + j] += first\_matrix[i \* n + k] \* second\_matrix[j \* n + k];

}

}

}

auto t2 = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> time = (t2 - t1);

std::cout << "Для матриц размером " << n << ": " << time.count() << " секунд" << std::endl;

return result\_matrix;

}

int main() {

setlocale(LC\_ALL, "Russian");

int n;

std::cout << "Введите размер матриц: ";

std::cin >> n;

auto matrix1 = multiplying(n);

auto matrix2 = transposed\_multiplying(n);

return 0;

}

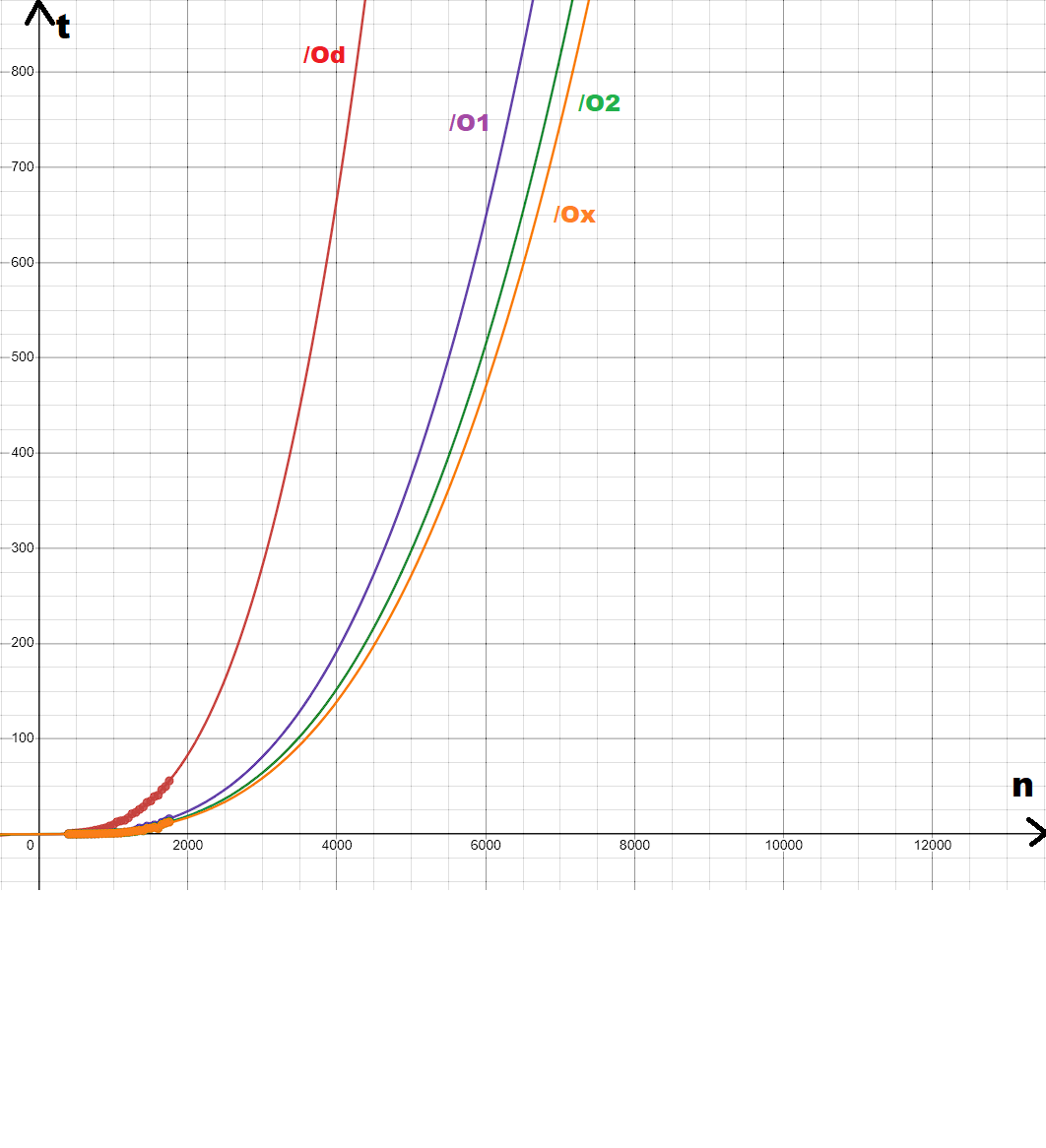
Ссылка на GitHub: <https://github.com/eternalowo/matrices-multiplying>

(Результаты замеров также лежат там в файле result.txt)

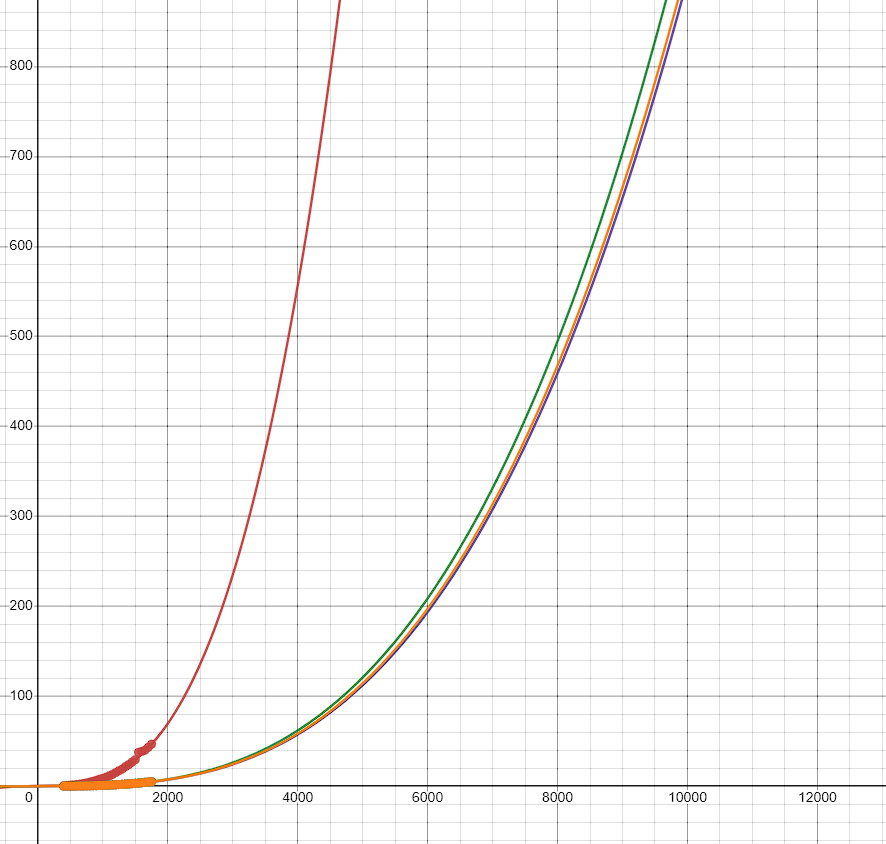
**График зависимости:**

График зависимости времени работы от размера входных данных (спрогнозирован и смоделирован на основе проведённых тестов (n от 400 до 1750)

Обычное перемножение:

****

Перемножение с транспонированной матрицей:

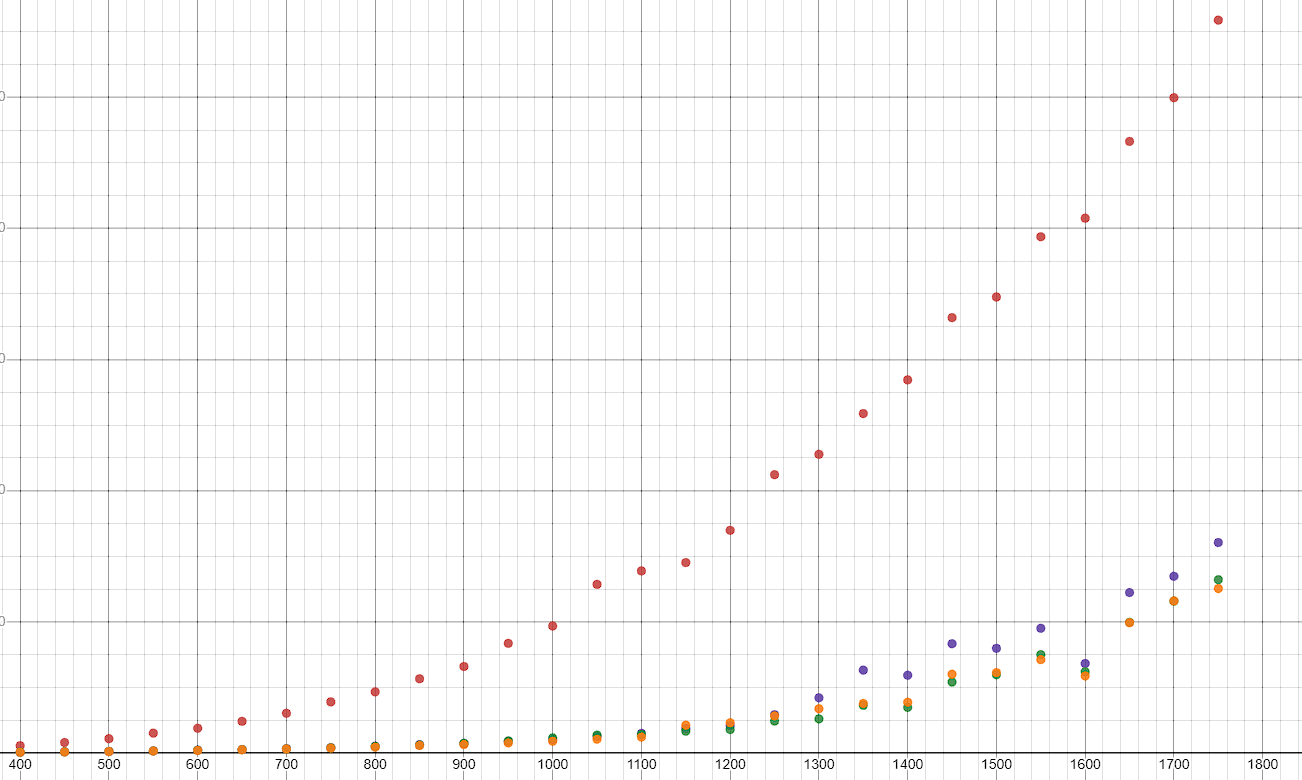


**Вывод:**

В данной работе были использованы одномерные векторы взамен двумерных дабы избежать кэш промахов и уменьшить время работы программы. Как мы видим, при разных уровнях оптимизации компилятора и способах – время работы программы сильно различается, разберем варианты по отдельности.

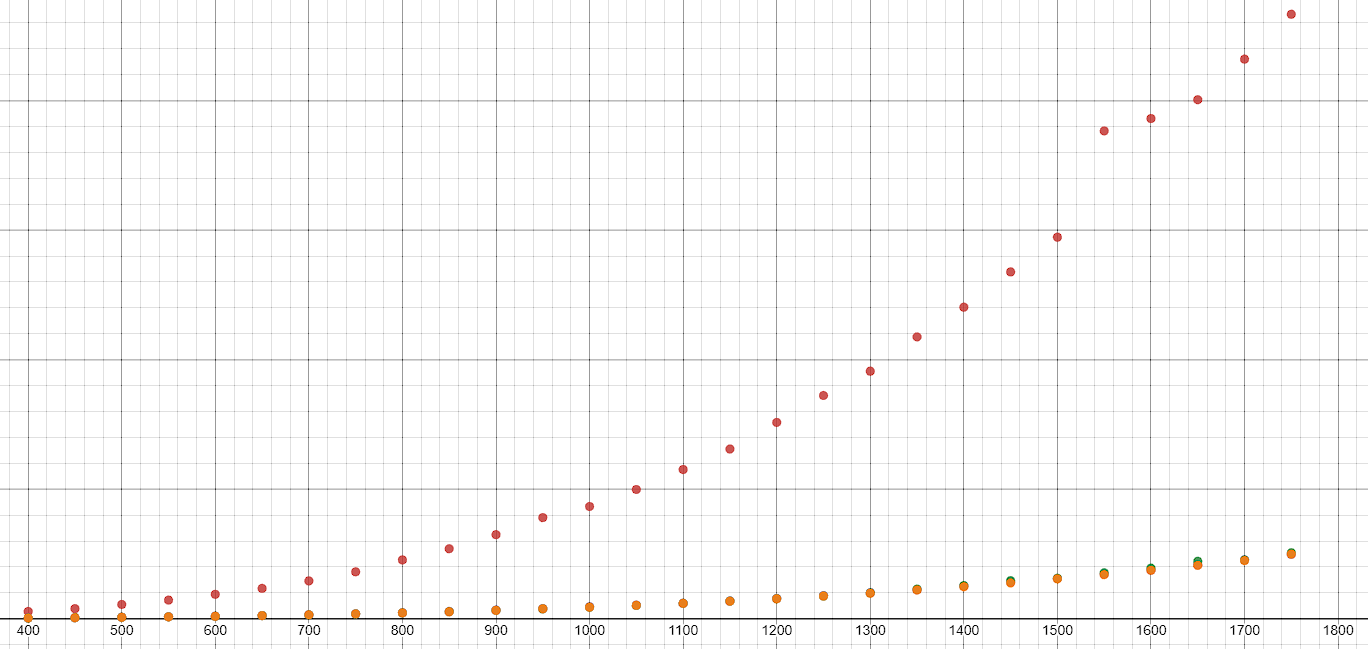
1. Обычное перемножение:

Значительно уступает отсутствие оптимизации (/Od) по сравнению с оставшимися тремя уровнями, /O1 при этом намного более приближен к /O2 и /Ox, можно сказать, что /O2 и /Ox в данном случае работают почти одинаково. При этом на уровнях оптимизации /O1, /O2 и /Ox программа сработала достаточно нестабильно, иногда на бОльших входных данных программа работала быстрее, чем на меньших (что особенно заметно на отметке n = 1600, пример на графике ниже, также /O1 показал самый нестабильный результат). Сложность программы, как и подтверждают графики – O(n3). (В ходе перемножения использовались 3 цикла от 0 до n)



1. Перемножение с транспонированной матрицей:

Отсутствие оптимизации также значительно уступает по сравнению с оставшимися тремя уровнями, причем они, при данном способе сработали практически одинаково (точки почти полностью совпадают) и намного стабильнее (пример на графике ниже) чем в первом варианте. Сложность программы, как и подтверждают графики, также как и в первом способе – O(n3).



Из спрогнозированных графиков и полученных результатов следует, что второй способ быстрее и намного стабильнее при различных уровнях оптимизации компилятора (за исключением её отсутствия (/Od), здесь ситуация между обоими способами почти аналогичная), поэтому при перемножении матриц будет намного лучше использовать построчный обход и любой из трёх уровней оптимизации - /O1, /O2, /Ox.