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: Хмелевцев Д.В.

Преподаватель: Перепёлкин В.А.

Новосибирск

2022

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

**Код**:

#include <vector>

#include <string>

#include <ctime>

#include <stdexcept>

#include <chrono>

#include <iostream>

#include <iomanip>

class MyMatrix

{

std::vector<float> matrix;

const size\_t size;

public:

MyMatrix(size\_t n) : size(n)

{

if (n > 1)

matrix = std::vector<float>(size \* size, 0);

else

throw std::invalid\_argument("Martix should have size at least 2x2\n");

}

size\_t get\_size() const { return size; }

float get\_element(const size\_t i, const size\_t j) const { return matrix[i \* size + j]; }

void fill\_randomly(int max\_value = 100, int accuracy = 0)

{

srand(time(NULL) % rand());

for (size\_t i = 0; i < size; i++)

for (size\_t j = 0; j < size; j++)

matrix[i \* size + j] =

float(std::rand() % int(max\_value \* pow(10, accuracy)) / (pow(10, accuracy)));

}

void transpose()

{

for (size\_t i = 0; i < size - 1; i++)

for (size\_t j = i + 1; j < size; j++)

std::swap(matrix[i \* size + j], matrix[j \* size + i]);

}

MyMatrix multiply(const MyMatrix& other) const

{

if (other.size == size)

{

MyMatrix result(size);

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

for (int i = 0; i < size; i++)

for (int j = 0; j < size; j++)

for (int k = 0; k < size; k++)

result.matrix[i \* size + j] +=

this->matrix[i \* size + k] \* other.matrix[k \* size + j];

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

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

std::cout << "Перемножение матриц размером " << size << ": "

<< time.count() << " секунд\n";

return result;

}

else

{

throw std::invalid\_argument("Martices must be the same size\n");

};

}

MyMatrix multiply\_transposed(const MyMatrix& other) const

{

if (other.size == size)

{

MyMatrix result(size);

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

for (int i = 0; i < size; i++)

for (int j = 0; j < size; j++)

for (int k = 0; k < size; k++)

result.matrix[i \* size + j] +=

this->matrix[i \* size + k] \* other.matrix[j \* size + k];

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

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

std::cout << "Перемножение матриц с транспонированием размером " << size << ": "

<< time.count() << " секунд\n";

return result;

}

else

{

throw std::invalid\_argument("Martices must be the same size\n");

};

}

friend std::ostream& operator<<(std::ostream& os, const MyMatrix& m)

{

os << std::setprecision(7);

os << "/" << std::string(m.size \* 13, ' ') << "\\\n";

for (size\_t i = 0; i < m.size; i++)

{

os << '|';

for (size\_t j = 0; j < m.size; j++)

os << ' ' << std::setw(11) << m.matrix[i \* m.size + j] << ' ';

os << "|\n";

}

os << "\\" << std::string(m.size \* 13, ' ') << "/\n\n";

return os;

}

};

int main()

{

setlocale(LC\_ALL, "");

std::vector<size\_t> sizes = {

512, 640, 768, 896, 1024, 1152, 1280, 1408, 1536, 1664, 1792, 1920, };

try

{

for (auto& size : sizes)

{

MyMatrix matrix1(size);

MyMatrix matrix2(size);

matrix1.fill\_randomly(100, 2);

matrix2.fill\_randomly(100, 2);

MyMatrix result1 = matrix1.multiply(matrix2);

matrix2.transpose();

MyMatrix result2 = matrix1.multiply\_transposed(matrix2);

}

}

catch (std::invalid\_argument ex) {

std::cout << ex.what();

return EXIT\_FAILURE;

}

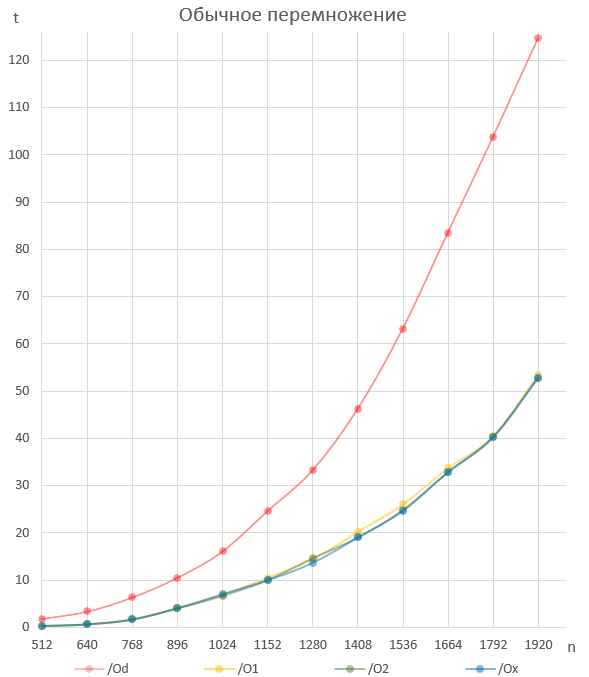
return EXIT\_SUCCESS;

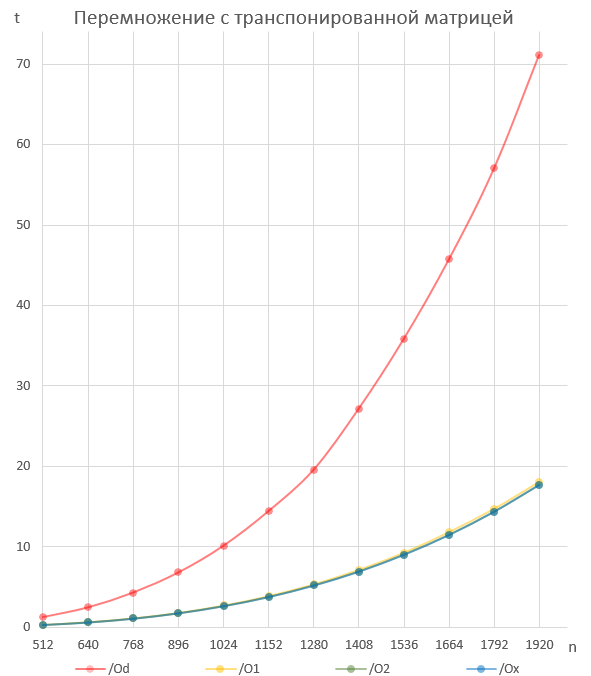
}

**Ссылка на GitHub**:

**Графики зависимости**:

Графики зависимости времени работы от размера входных данных (n от 512 до 1920)

****

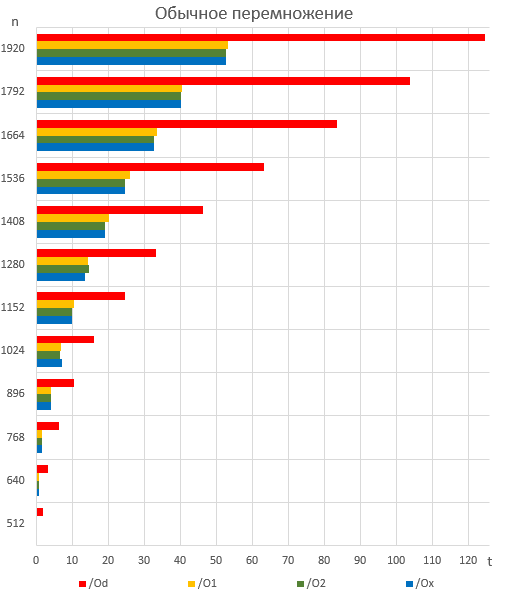


**Вывод:**

В работе были использованы одномерные векторы вместо двумерных, чтобы, используя принцип локальности данных, избежать кэш промахов и уменьшить время работы программы. На разных уровнях оптимизации компилятора и способах – время работы программы сильно различается, разберем каждый вариант.

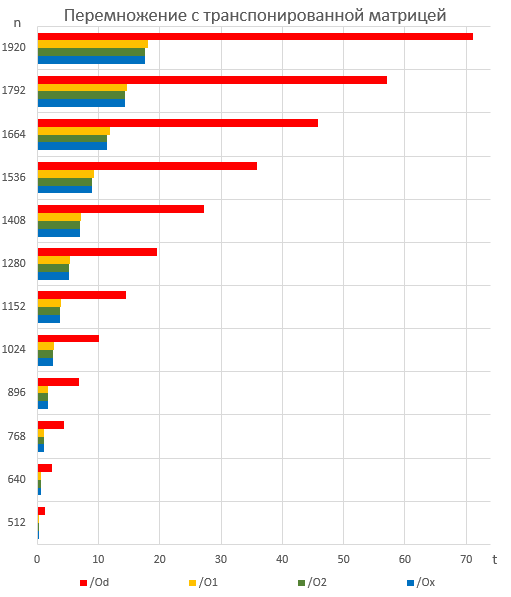
Обычное перемножение:

Значительно уступает с отсутствием оптимизации (/Od) по сравнению с оставшимися тремя уровнями, /O1 приближен к /O2 и /Ox, при этом можно сказать, что /O2 и /Ox в данном случае работают почти одинаково. Сложность программы – O(n3) (3 цикла от 0 до n).



Перемножение с транспонированной матрицей:

Отсутствие оптимизации также значительно уступает по времени оставшимся трем уровням, 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 полученных результатов следует, что второй способ быстрее и намного стабильнее при различных уровнях оптимизации компилятора (за исключением её отсутствия (/Od), поэтому при перемножении матриц будет намного лучше использовать построчный обход и любой из трёх уровней оптимизации - /O1, /O2, /Ox.