###### МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

###### ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

###### НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

###### Факультет информационных технологий

**Кафедра параллельных вычислений**

**ОТЧЕТ**

**О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ**

«ВЛИЯНИЕ КЭШ-ПАМЯТИ НА ВРЕМЯ ОБРАБОТКИ МАССИВОВ»

студента 2 курса, группы 22208

Новикова Григория Андреевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Антон Юрьевич Кудинов

Новосибирск 2023

1. **Цель**

Изучить принцип работы работа кэш-память процессора и на примере обхода элементов массива в разном порядке выяснить размер разных уровней кэш-памяти на конкретном ПК.

1. **Задание**

На основе разных способов обхода циклического массива: прямого, обратного и случайного, сделать среднюю оценку количества тактов для обращения к одному элементу массива. На основе полученных данных выяснить приблизительный размер кэша процессора.

1. **Описание обходов**

Используются следующие обходы массива:

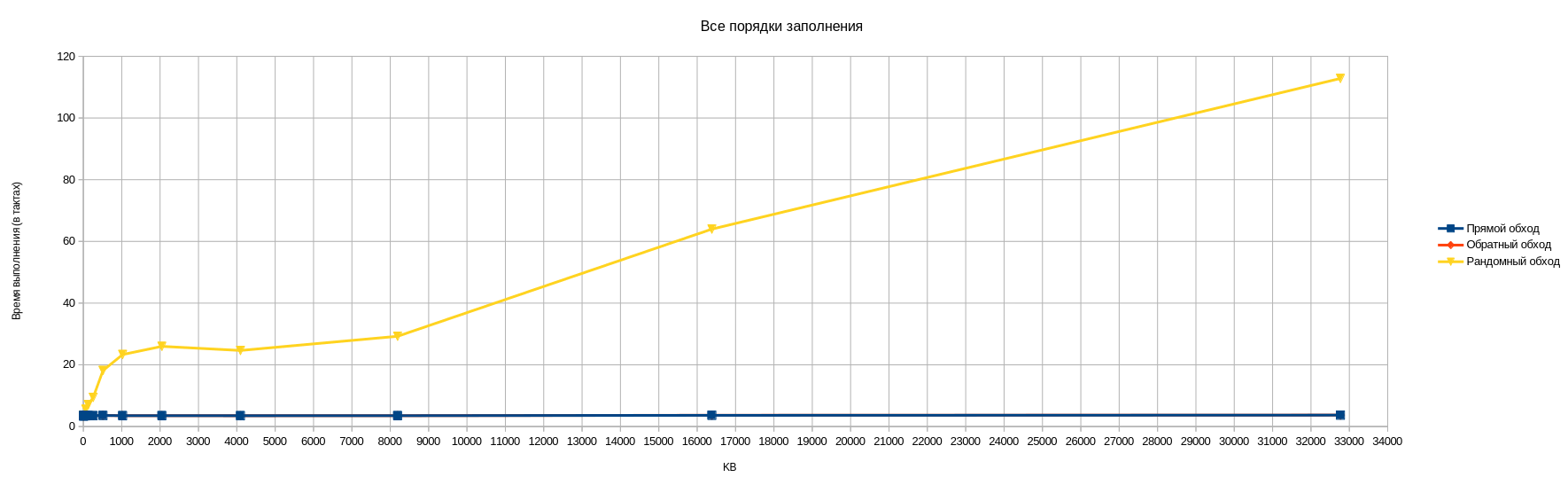
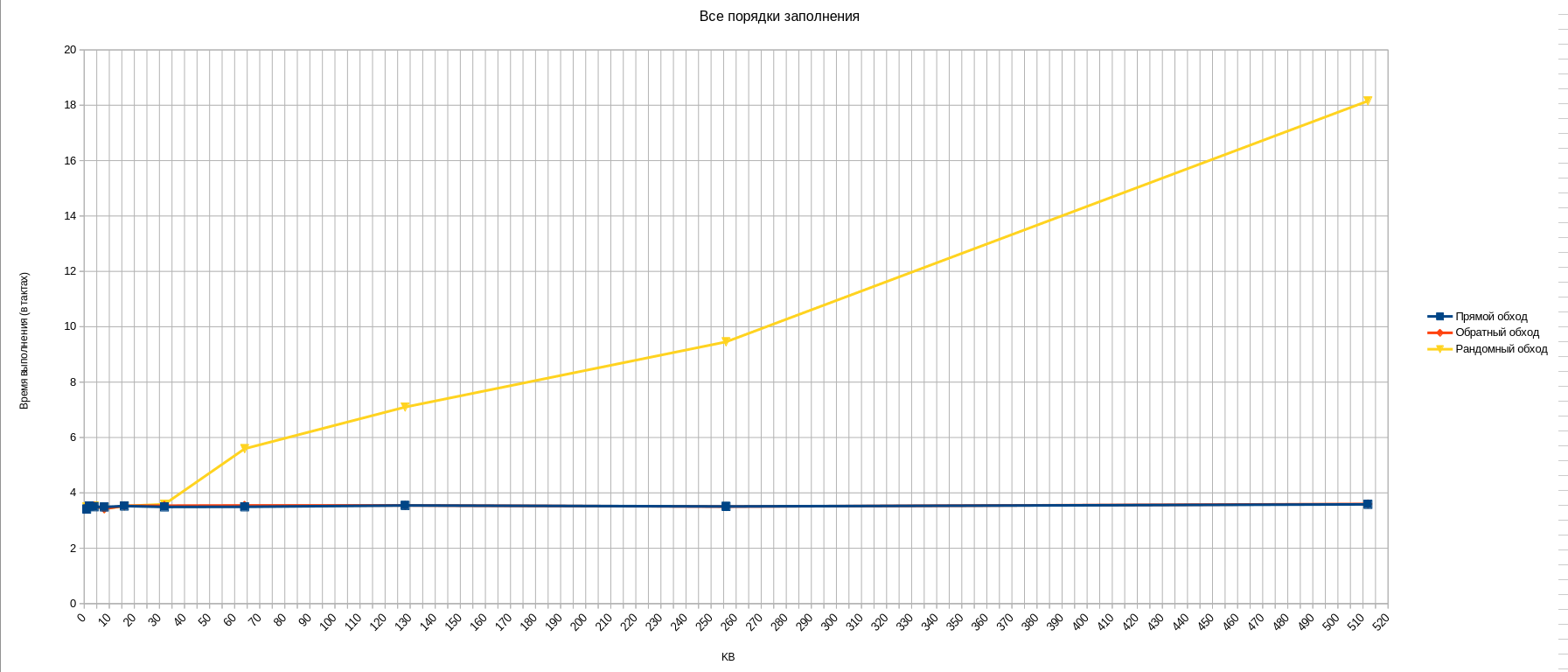
* Прямой;
* Обратный;
* Случайный.

**Прямой обход** подразумевает, что значением ячейки массива с индексом i будет i+1, т.е. индекс следующей соседней ячейки массива, в случае последнего элемента следующим элементом будет самый первый. Таким образом мы получаем следующий индекс и так обходим массив несколько раз.

**Обратный обход** – прямой обход, но с условием, что теперь обход идёт с конца массива линейным порядком.

**Случайный обход** подразумевает, что значением ячейки массива с индексом i будет некоторый индекс, который невозможно предугадать, но гарантируется, что индекс не выйдет за пределы массива. Тем самым, ходя по разным индексам, мы полностью обходим массив.

1. **График роста тактов обращения**



1. **Листинг программы**

#include <iostream>

#include <x86intrin.h>

#include <float.h>

using namespace std;

#define K 10

int Run(int \*arr, int size, int countRun) {

int k, i;

for (k = 0, i = 0; i < size \* countRun; i++) {

k = arr[k];

}

return k;

}

int \*CreateStraight(int size) {

int \*arr = new int[size];

arr[size - 1] = 0;

for (int i = 1; i < size; i++) {

arr[i - 1] = i;

}

return arr;

}

int \*CreateReverse(int size) {

int \*arr = new int[size];

arr[0] = size - 1;

for (int i = 1; i < size; i++) {

arr[i] = i - 1;

}

return arr;

}

int \*CreateRandom(int size) {

int \*arr = new int[size];

auto \*indexes = new unsigned int[size];

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

indexes[i] = i;

}

srand(time(NULL));

for (long long i = size - 1; i > 0; --i) {

unsigned int j = 0;

j = rand() % i;

swap(indexes[i], indexes[j]);

}

for (unsigned int i = 0; i < size - 1; ++i) {

arr[indexes[i]] = indexes[i + 1];

}

arr[indexes[size - 1]] = indexes[0];

delete[] indexes;

return arr;

}

int main()

{

int size[16] = {

1 \* 1024 / 4, // 1 KB

2 \* 1024 / 4, // 2 KB

4 \* 1024 / 4, // 4 KB

8 \* 1024 / 4, // 8 KB

16 \* 1024 / 4, // 16 KB

32 \* 1024 / 4, // 32 KB

64 \* 1024 / 4, // 64 KB

128 \* 1024 / 4, // 128 KB

256 \* 1024 / 4, // 256 KB

512 \* 1024 / 4, // 512 KB

1 \* 1024 \* 1024 / 4, // 1 MB

2 \* 1024 \* 1024 / 4, // 2 MB

4 \* 1024 \* 1024 / 4, // 4 MB

8 \* 1024 \* 1024 / 4, // 8 MB

16 \* 1024 \* 1024 / 4, // 16 MB

32 \* 1024 \* 1024 / 4 // 32 MB

};

int \*arr1;

int \*arr2;

int \*arr3;

double minTime = DBL\_MAX;

int k = 0;

for (int i = 0; i < 16; i++) {

std::cout << "===============" << std::endl;

std::cout << i + 1 << " Memory size in KB: " << size[i] \* 4 / 1024 << std::endl;

arr1 = CreateStraight(size[i]);

k += Run(arr1, size[i], 1);

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

double start = \_\_rdtsc();

k += Run(arr1, size[i], K);

double end = \_\_rdtsc();

minTime = minTime > ((double)(end - start)) ?

((double)(end - start)) : minTime;

}

delete [] arr1;

minTime /= K;

minTime /= size[i];

std::cout << " Time for straight: " << minTime << std::endl;

minTime = DBL\_MAX;

arr2 = CreateReverse(size[i]);

k += Run(arr2, size[i], 1);

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

double start = \_\_rdtsc();

k += Run(arr2, size[i], K);

double end = \_\_rdtsc();

minTime = minTime > ((double)(end - start)) ?

((double)(end - start)) : minTime;

}

delete [] arr2;

minTime /= K;

minTime /= size[i];

std::cout << " Time for reverse: " << minTime << std::endl;

minTime = DBL\_MAX;

arr3 = CreateRandom(size[i]);

k += Run(arr3, size[i], 1);

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

double start = \_\_rdtsc();

k += Run(arr3, size[i], K);

double end = \_\_rdtsc();

minTime = minTime > ((double)(end - start)) ?

((double)(end - start)) : minTime;

}

delete [] arr3;

minTime /= K;

minTime /= size[i];

std::cout << " Time for random: " << minTime << std::endl

<< "===============" << std::endl;

minTime = DBL\_MAX;

}

std::cout << k;

return 0;

}

**Команда для компиляции:**

**g++ -O1 main.cpp**

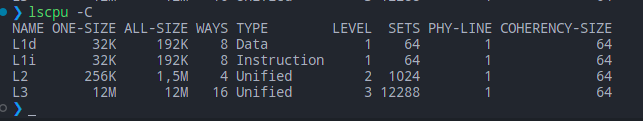
1. **Оценка размера кэша**

По графику можно заметить, что прирост тактов начинается с 32kB, что явно говорит о том, что размер L1 кэша равен 32kB.

Следующее возрастание между 230Kb и 290Kb, что соответствует размеру L2 кэша - 256Kb.

В следующий раз более сильный прирост можно заметить между 8000Kb и 16000Kb, что очень похоже на размер L3 кэша = 12MB.

Данные, полученные c помощью команды lscpu подтверждают данные предположения.



Вследствие предвыборки данных, прямой и обратный обходы не росли с увеличением размера массива. Зато видна особенность обхода массива в случайном порядке: в этом случае количество тактов процессора, затраченное на получение элемента массива, заметно растет с увеличением размера массива потому что что кэш-контроллер не может корректно выполнить предвыборку данных для обхода массива в случайном порядке.

1. **Заключение**

В ходе данной работы были установлены размеры кэшей процессора i5-10500H и выяснено, насколько существенен прирост времени обращения при "переходе" с одного кэша на другой.