

**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 ПРАКТИЧЕСКОЙ РАБОТЫ

«Изучение быстродействия кэша процессора»

студента 2 курса, группы 20205

Муратова Максима Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

**Преподаватель:
Доцент
Власенко А. Ю.**

Новосибирск 2021

СОДЕРЖАНИЕ

ЦЕЛИ.....	3
ЗАДАНИЕ.....	3
ОПИСАНИЕ РАБОТЫ.....	4
ЗАКЛЮЧЕНИЕ.....	6
Приложение 1. <i>main.cpp</i>	9
Приложение 2. <i>mtrx.h</i>	11
Приложение 3. <i>mtrx.cpp</i>	12
Приложение 4. <i>direct.cpp</i>	14
Приложение 5. <i>reversed.cpp</i>	15
Приложение 6. <i>pseudorandom.cpp</i>	16
Приложение 7. <i>random.cpp</i>	18
Приложение 8. <i>create.sh</i>	19
Приложение 9. <i>test.sh</i>	20
Приложение 10. <i>Полезные команды</i>	23
Приложение 11. <i>Таблица к измерению скорости обходов</i>	24

ЦЕЛИ

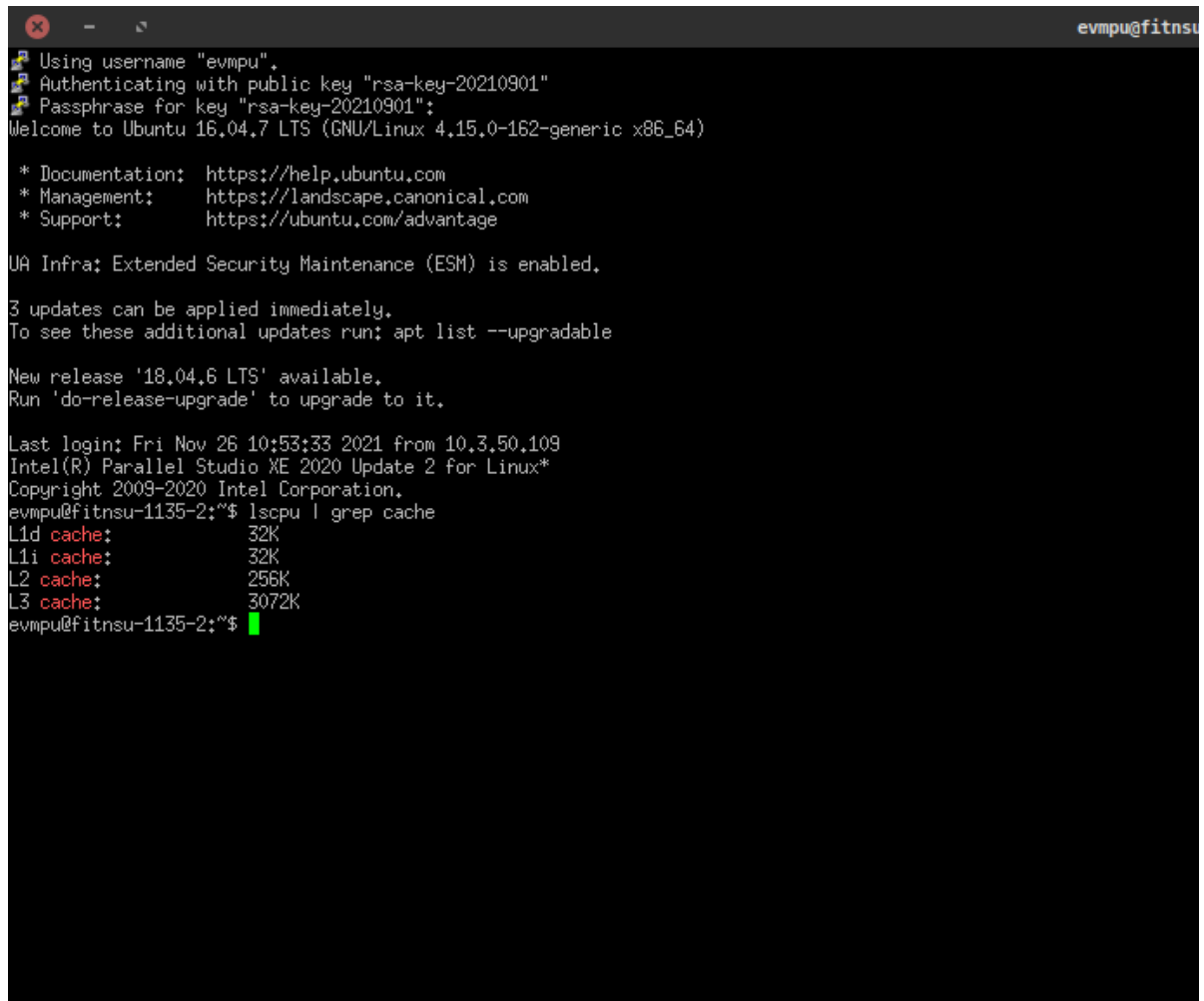
- Узнать размеры своего кэша
- Узнать, как сильно способ обхода массива влияет на скорость самого обхода и обосновать это

ЗАДАНИЕ

1. Узнать размер кэша при помощи системных утилит
2. Написать программу, которая совершает проход массив 4 разными способами: прямым, обратным, почти случайным и случайным
3. По графику зависимости скорости случайного обхода массива экспериментально узнать размеры кэша и сравнить это с реальными значениями

ОПИСАНИЕ РАБОТЫ

Для начала узнаем размер нашего кэша. Для этого используем команду
`$ lscpu | grep cache`



```
evmpu@fitnsu
Using username "evmpu".
Authenticating with public key "rsa-key-20210901"
Passphrase for key "rsa-key-20210901":
Welcome to Ubuntu 16.04.7 LTS (GNU/Linux 4.15.0-162-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

UA Infra: Extended Security Maintenance (ESM) is enabled.

3 updates can be applied immediately.
To see these additional updates run: apt list --upgradable

New release '18.04.6 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

Last login: Fri Nov 26 10:53:33 2021 from 10.3.50.109
Intel(R) Parallel Studio XE 2020 Update 2 for Linux*
Copyright 2009-2020 Intel Corporation.
evmpu@fitnsu-1135-2:~$ lscpu | grep cache
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            3072K
evmpu@fitnsu-1135-2:~$
```

(L1i для инструкций, его в расчёт не берём)

Если это перевести это в переменные типа `int`, то есть в блоки по 4 байта, то имеем такие размеры:

1. L1d – 8192 переменные типа `int`
2. L2 – 65 536 переменных типа `int`, вместе с предыдущим 73 728 переменных типа `int`
3. L3 – 786 432 переменные типа `int`, вместе с предыдущими 860 160 переменных типа `int`

Теперь напишем саму программу для разных обходов массива

Для удобства составления программы она была поделена на 6 частей:

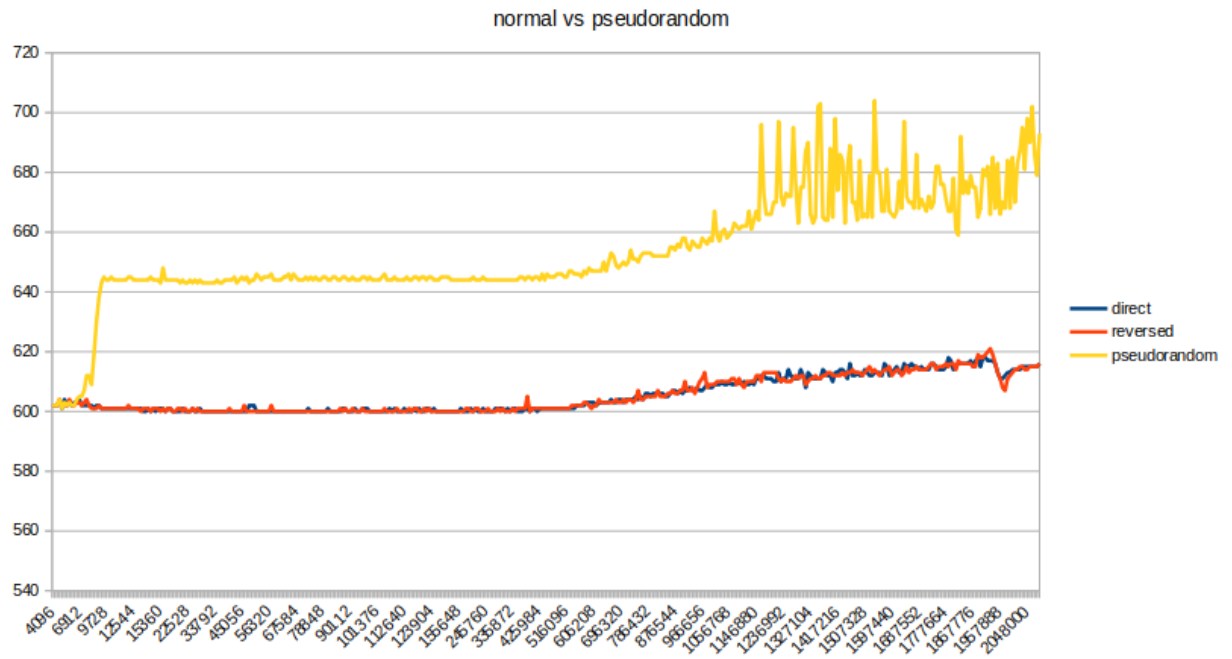
1. `main.cpp` – собственно реализация алгоритма обхода массива (ПРИЛОЖЕНИЕ 1)

2. `mtrx.h`, `mtrx.cpp` – класс `Matrix`, с помощью которого разгонялся процессор, позаимствован из Работы по векторизации (ПРИЛОЖЕНИЯ 2 и 3 соответственно)
3. `traversal.h`, `direct.cpp`, `reversed.cpp`, `pseudorandom.cpp`, `random.cpp` – реализация функции всех четырёх обходов массива (ПРИЛОЖЕНИЯ 4-7 соответственно)

Коротко об использованных обходах

1. Прямой – проходим нулевой, первый, второй, ..., $n-1$, нулевой элементы
2. Обратный – то же, что и прямой, но в обратную сторону
3. Псевдослучайный (изначально просто случайный) – следующим способом:
 1. Берём один с левого края
 2. Берём один с правого
 3. Два из центра справа налево
 4. Один с левого края
 5. Два из центра слева направо
 6. Один с правого края
 7. Если ещё не все элементы использованы, см. п. 1
 8. Иначе после последнего идём в 0
4. Случайный – при помощи `std::random_shuffle()`

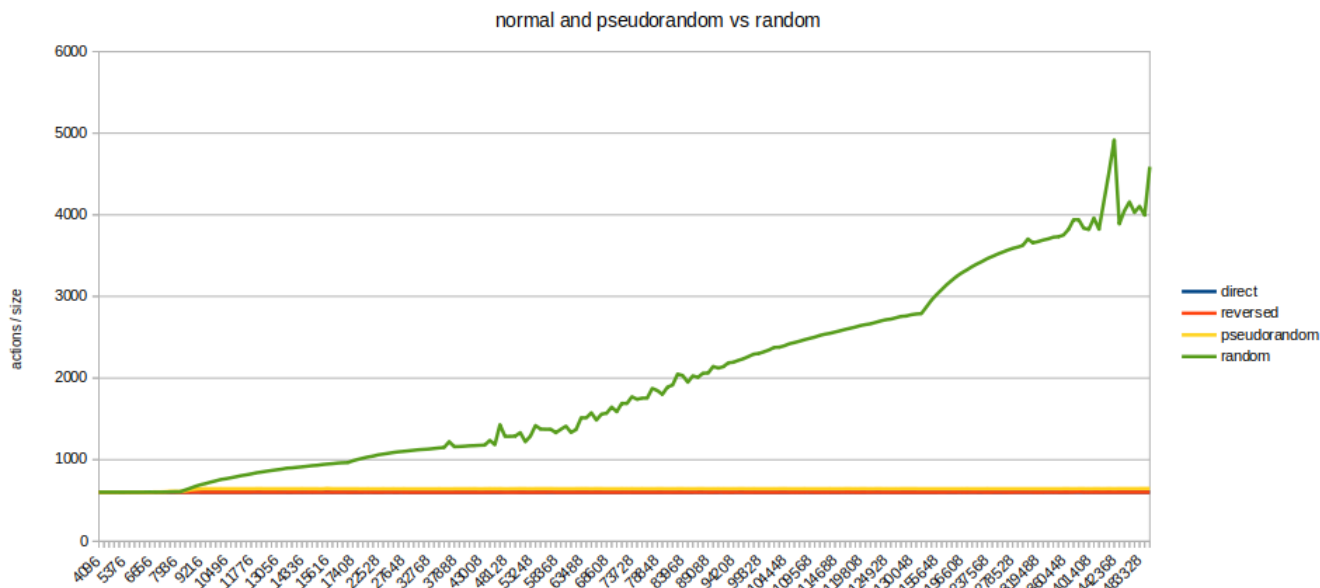
Сначала я протестировал дважды первые 3 способа, из двух вариантов я выбрал наименьший (это также плюс к шумоподавлению) и получил следующую картину:



По Ох – размер массива в int, по Оу – число тактов, необходимое на доступ к одному элементу для всех прогонов, которых везде было 100. Специально не делил на всё, чтобы были детали, а они есть, и вот какие:

- На размере массива 8448 переменных типа int (33 КБ) время на псевдослучайный обход взлетает, но не так сильно, как это ожидалось. Это потом сподвигло меня написать «настоящий» рандом. Тем не менее, отсюда можно извлечь важный микровывод: кэш используется менее эффективно при «почти» рандоме
- На размере массива 581632 (2272 КБ) переменных типа int начинают понемногу расти и прямой и обратный обходы, что сигнализирует о том, что и им в какой-то момент не хватило кэша L1d (L1i только для инструкций, не для данных), но рост у них не такой стремительный, как у настоящего рандома

Как я уже отметил, мой первоначальный рандом не так сильно проигрывает прямому и обратному обходам, поэтому я написал рандом с гораздо большим разбросом. Протестировал я его так же, как и первые три обхода – лучший из двух. Результаты прогонок я отобразил на следующем графике:



Оси тут точно такие же

Отсюда можно извлечь следующее:

1. При размере массива 8448 (33 КБ) переменных типа `int` время на рандом первый раз резко взлетает, что означает, что ему не хватило L1d
2. При размере массива 17408 (68 КБ) интов не хватает уже и L2, так что мы видим второй резкий рост
3. При размере массива 139264 (544 КБ) интов обход не помещается и в L3, что означает, что теперь оперативная память используется напрямую, о чём и сигнализирует ещё более крутой график
4. Как можно заметить, это никак не вяжется с настоящими размерами кэша. Причины могут быть таковы:
 1. Процессор дёргает регулярно не только наш массив, но и другие данные, что не даёт использовать L1d полностью под наш массив
 2. L3 используется несколькими ядрами

Напомню, что кэш хранит именно участок данных, а не одну ячейку, из предположения, что если было обращение к памяти, то скоро будет обращение к памяти к соседней ячейке. Это, скорее всего, главная причина, по которой случайный обход проигрывает прямому, обратному и даже псевдослучайному, всё из-за разной эффективности использования кэша.

Табличные данные представлены в ПРИЛОЖЕНИИ 11.

ЗАКЛЮЧЕНИЕ

1. При использовании прямого и обратного обхода число тактов на доступ к одной переменной почти постоянен, потому что кэш загружает именно участок памяти
2. Процессор не сможет отдать нам весь кэш, потому что ему нужны и другие данные, особенно если речь об другом ядре
3. Если алгоритм плохо рандомизирован, то производительность если и пострадает, то не так сильно, как при совсем случайном обходе, где кэш используется наименее неэффективно

Приложение 1. *main.cpp* основной текст программы

```
#include <iostream>
#include <fstream>
#include "mtrx.h"
#include "traversal.h"

int main(int argc, char **argv) {
    if(argc != 3) {
        std::cerr << "Wrong input\n";
        return 0;
    }

    union ticks {
        unsigned long long t64 = 0;
        struct s32 { long th, tl; } t32;
    } start, end;

    int size = atoi(argv[1]), watchingCount = atoi(argv[2]);
    int *arr = new int[size];
    createLoop(arr, size);

    //acceleration
    auto *a = new Matrix{10}, *b = new Matrix{*a};
    for(int index = 0; index < 410000; index++)
        *a *= *b;
    delete a;
    delete b;

    std::ofstream out("/dev/null");
```

```

//heating
int pos = 0;
for(int index = 0; index < size * watchingCount; index++) {
    pos = arr[pos];
    if(pos == 314) out << "100 Pi\n";
}

asm("rdtsc\n":"=a"(start.t32.th),"=d"(start.t32.tl));
for(int index = 0; index < size * watchingCount; index++) {
    pos = arr[pos];
    if(pos == 314) out << "100 Pi\n";
}
asm("rdtsc\n":"=a"(end.t32.th),"=d"(end.t32.tl));

std::cout << size << '\t' << (end.t64 - start.t64) /
    (unsigned long long) size << '\n';

out.close();
return 0;
}

```

Приложение 2. *mtrx.h*
заголовочный файл класса *Matrix*

```
#ifndef VECTOR_MTRX_H
#define VECTOR_MTRX_H

#include <iostream>

class Matrix {
    int size;
    float *data;
public:
    explicit Matrix(int length);
    ~Matrix();
    Matrix(const Matrix &otherMatrix);

    Matrix &operator*=(const Matrix &b);
};

#endif //VECTOR_MTRX_H
```

Приложение 3. *mtrx.cpp*

реализация основных методов класса *Matrix*

```
#include "mtrx.h"
```

```
Matrix::Matrix(int length): size(length) {  
    data = new float[size * size];  
    for(int index = 0; index < size; index++) {  
        data[index * (size + 1)] = 1;  
    }  
}
```

```
Matrix::~~Matrix() {  
    delete[] data;  
    data = nullptr;  
}
```

```
Matrix::Matrix(const Matrix &otherMatrix):  
    size(otherMatrix.size)  
{  
    data = new float[size * size];  
    for(int index = 0; index < size * size; index++)  
        data[index] = otherMatrix.data[index];  
}
```

```
Matrix &Matrix::operator*=(const Matrix &b) {  
    if(size == b.size) {  
        auto *newData = new float[size * size];  
        if(size == 4) newData[4] = 0; //else I will get a nan  
        for(int x = 0; x < size; x++) {  
            for(int y = 0; y < size; y++) {  
                for(int i = 0; i < size; i++)
```

```
        newData[x * size + y] += data[x * size + i]
    * b.data[i * size + y];
    }
}
delete[] data;
data = newData;
}
return *this;
}
```

Приложение 4. *direct.cpp*
реализация прямого обхода массива

```
#include "traversal.h"

int *createLoop(int size) {
    for(int index = 0; index < size - 1; index++)
        arr[index] = index + 1;
    return arr;
}
```

Приложение 5. *reversed.cpp*
реализация обратного обхода массива

```
#include "traversal.h"

void createLoop(int *arr, int size) {
    arr[0] = size - 1;
    for(int index = 2; index < size; index++)
        arr[index] = index - 1;
    return arr;
}
```

Приложение 6. *pseudorandom.cpp*
реализация псевдослучайного обхода массива

```
#include "traversal.h"

void createLoop(int *arr, int size) {
    int leftEdge = 0, leftCenter = (size - 1) / 2,
    rightCenter = leftCenter + 1, rightEdge = size - 1;

    while(leftEdge <= leftCenter) {
        if(rightCenter <= rightEdge) {
            arr[leftEdge] = rightEdge;
            leftEdge++;
        }

        if(rightCenter <= rightEdge) {
            arr[rightEdge] = rightCenter;
            rightEdge--;
        }

        if(leftEdge <= leftCenter) {
            arr[rightCenter] = leftCenter;
            rightCenter++;
        }

        if(leftEdge <= leftCenter) {
            arr[leftCenter] = leftEdge;
            leftCenter--;
        }

        if(leftEdge <= leftCenter) {
            arr[leftEdge] = leftCenter;
            leftEdge++;
        }
    }
}
```



```

    }

    if(rightCenter <= rightEdge) {
        arr[leftCenter] = rightCenter;
        leftCenter--;
    }

    if(rightCenter <= rightEdge) {
        arr[rightCenter] = rightEdge;
        rightCenter++;
    }

    if(leftEdge <= leftCenter) {
        arr[rightEdge] = leftEdge;
        rightEdge--;
    }
}

if(arr[leftEdge] == leftEdge) arr[leftEdge] = 0;
if(arr[leftCenter] == leftCenter) arr[leftCenter] = 0;
if(arr[rightCenter] == rightCenter) arr[rightCenter] = 0;
if(arr[rightEdge] == rightEdge) arr[rightEdge] = 0;

return arr;
}

```

Приложение 7. *random.cpp*
реализация случайного обхода массива

```
#include "traversal.h"
#include <algorithm>

void createLoop(int *arr, int size) {
    int *order = new int[size];
    for(int index = 0; index < size - 1; index++)
        order[index] = index + 1;
    std::random_shuffle(order, order + size - 1);

    int pos = 0;
    for(int index = 0; index < size; index++) {
        arr[pos] = order[index];
        pos = arr[pos];
    }

    delete[] order;
    return arr;
}
```

Приложение 8. *create.sh*

```
#!/bin/bash

g++ -O1 -std=c++11 main.cpp mtrx.h mtrx.cpp
    traversal.h direct.cpp -o prog-direct

g++ -O1 -std=c++11 main.cpp mtrx.h mtrx.cpp
    traversal.h reversed.cpp -o prog-reversed

g++ -O1 -std=c++11 main.cpp mtrx.h mtrx.cpp
    traversal.h pseudorandom.cpp -o prog-pseudorandom

g++ -O1 -std=c++11 main.cpp mtrx.h mtrx.cpp
    traversal.h random.cpp -o prog-random
```

Приложение 9. *test.sh*

```
#!/bin/bash

echo "Direct test"

echo "part1"
for((size = 4096 ; size < 16385 ; size = $size + 256 )); do
    ./prog-direct $size 100 | tee -a
        report-direct.txt >/dev/null
done

echo "part2"
for((size = 16384 ; size < 131073 ; size = $size + 1024 ));
do
    ./prog-direct $size 100 | tee -a
        report-direct.txt >/dev/null
done

echo "part3"
for((size = 131072 ; size < 2097152 ; size = $size + 8192 ));
do
    ./prog-direct $size 100 | tee -a
        report-direct.txt >/dev/null
done

echo "Reversed test"

echo "part1"
for((size = 4096 ; size < 16385 ; size = $size + 256 )); do
    ./prog-reversed $size 100 | tee -a
        report-reversed.txt >/dev/null
done
```

```

echo "part2"
for((size = 16384 ; size < 131073 ; size = $size + 1024 ));
do
    ./prog-reversed $size 100 | tee -a
        report-reversed.txt >/dev/null
done

echo "part3"
for((size = 131072 ; size < 2097152 ; size = $size + 8192 ));
do
    ./prog-reversed $size 100 | tee -a
        report-reversed.txt >/dev/null
done

echo "Pseudorandom test"

echo "part1"
for((size = 4096 ; size < 16385 ; size = $size + 256 )); do
    ./prog-pseudorandom $size 100 | tee -a
        report-pseudorandom.txt >/dev/null
done

echo "part2"
for((size = 16384 ; size < 131073 ; size = $size + 1024 ));
do
    ./prog-pseudorandom $size 100 | tee -a
        report-pseudorandom.txt >/dev/null
done

echo "part3"
for((size = 131072 ; size < 2097152 ; size = $size + 8192 ));
do
    ./prog-pseudorandom $size 100 | tee -a
        report-pseudorandom.txt >/dev/null
done

```

```
echo "Random test"

echo "part1"
for((size = 4096 ; size < 16385 ; size = $size + 256 )); do
    ./prog-random $size 100 | tee -a
        report-random.txt >/dev/null
done

echo "part2"
for((size = 16384 ; size < 131073 ; size = $size + 1024 ));
do
    ./prog-random $size 100 | tee -a
        report-random.txt >/dev/null
done

echo "part3"
for((size = 131072 ; size < 2097152 ; size = $size + 8192 ));
do
    ./prog-random $size 100 | tee -a
        report-random.txt >/dev/null
done

unset size
```

Приложение 10. Полезные команды

- Копирование репозитория
`$ git clone https://github.com/mu2so4/evmpu-lab6`
- Обновление данных в локальной директории
`$ cd 20205/Muratov/evmpu-lab6`
`$ git pull origin master`
- Фиксация изменений файла
`$ git add <filename>`
- Коммит
`$ git commit -m "Some commit message"`
- Загрузка изменений в облачную репозиторию (нужен токен)
`$ git push`
- Сборка исполняемых файлов
`$./create.sh`
- Замер времени работы программ с записью результатов в `report.txt`
`$./test.sh`

Приложение 11. Зависимость числа тактов от размера массива и способа его обхода

Число обходов N = 100

1 КБ = 256 переменных типа int

Размер массива, число переменных int	Среднее время доступа к элементу при прямом обходе, такты * число обходов	Среднее время доступа к элементу при обратном обходе, такты * число обходов	Среднее время доступа к элементу при псевдослучайном обходе, такты * число обходов	Среднее время доступа к элементу при случайном обходе, такты * число обходов
4096	602	602	602	602
4352	602	602	602	602
4608	602	602	602	602
4864	603	604	604	602
5120	601	602	601	601
5376	604	602	603	601
5632	602	602	602	602
5888	603	604	603	602
6144	602	602	602	602
6400	603	602	602	602
6656	603	603	604	604
6912	604	603	605	602
7168	602	602	605	603
7424	602	603	607	604
7680	602	604	612	605
7936	602	602	612	607
8192	602	601	609	607
8448	601	601	619	629
8704	602	601	630	650
8960	602	602	638	673
9216	601	601	643	693
9472	601	601	645	707

9728	601	601	644	725
9984	601	601	644	740
10240	601	601	645	757
10496	601	601	644	766
10752	601	601	644	778
11008	601	601	644	790
11264	601	601	644	804
11520	601	601	644	814
11776	601	601	644	825
12032	601	602	645	839
12288	601	601	645	848
12544	601	601	644	858
12800	601	601	644	867
13056	601	601	644	876
13312	601	600	644	884
13568	600	601	644	896
13824	600	601	644	899
14080	601	601	644	906
14336	600	600	645	912
14592	601	601	644	919
14848	600	601	644	927
15104	601	601	644	931
15360	601	600	643	940
15616	601	601	648	946
15872	600	600	644	951
16128	601	601	644	958
16384	601	601	644	962
16384	600	600	644	963
17408	600	600	644	986
18432	600	601	644	1003
19456	600	601	643	1018
20480	601	600	644	1033

21504	600	601	643	1043
22528	600	600	643	1059
23552	600	600	644	1068
24576	601	601	643	1078
25600	600	600	644	1088
26624	600	601	643	1096
27648	601	600	644	1102
28672	600	600	643	1108
29696	600	600	643	1115
30720	600	600	643	1122
31744	600	600	643	1126
32768	600	600	643	1130
33792	600	600	643	1137
34816	600	600	644	1144
35840	600	600	643	1149
36864	600	600	643	1221
37888	600	600	644	1159
38912	600	600	644	1161
39936	600	601	644	1165
40960	600	600	644	1170
41984	600	600	645	1172
43008	600	600	643	1176
44032	600	600	644	1178
45056	600	600	645	1236
46080	600	602	644	1184
47104	600	600	645	1428
48128	602	600	643	1285
49152	602	600	644	1286
50176	602	600	644	1288
51200	600	600	646	1331
52224	600	600	645	1220
53248	600	600	644	1288

54272	600	600	645	1418
55296	600	600	645	1375
56320	600	600	645	1372
57344	600	602	646	1373
58368	600	600	644	1330
59392	600	600	644	1372
60416	600	600	644	1410
61440	600	600	644	1333
62464	600	600	645	1369
63488	600	600	645	1514
64512	600	600	646	1512
65536	600	600	644	1574
66560	600	600	646	1485
67584	600	600	645	1558
68608	600	600	644	1569
69632	600	600	644	1644
70656	600	600	644	1588
71680	600	600	645	1690
72704	601	600	644	1688
73728	600	600	645	1771
74752	600	600	644	1740
75776	600	600	645	1754
76800	600	600	644	1754
77824	600	600	644	1873
78848	600	600	645	1847
79872	600	600	645	1800
80896	601	600	644	1890
81920	600	600	644	1914
82944	600	600	645	2049
83968	600	600	645	2031
84992	600	600	644	1949
86016	600	601	644	2027

87040	601	600	645	2008
88064	601	601	645	2060
89088	600	600	644	2060
90112	600	600	644	2142
91136	600	601	645	2122
92160	601	600	644	2139
93184	600	600	644	2187
94208	600	600	644	2194
95232	601	601	645	2218
96256	601	600	645	2237
97280	601	600	644	2265
98304	600	600	645	2293
99328	600	600	644	2302
100352	600	600	644	2321
101376	600	600	644	2344
102400	600	600	644	2375
103424	600	600	645	2378
104448	600	601	646	2395
105472	600	600	644	2420
106496	601	600	644	2435
107520	600	600	644	2451
108544	601	600	645	2470
109568	600	601	644	2486
110592	600	600	644	2501
111616	600	600	644	2522
112640	601	600	644	2537
113664	600	600	645	2548
114688	600	601	644	2564
115712	600	600	644	2579
116736	601	601	645	2596
117760	600	601	645	2610
118784	601	600	644	2624

119808	600	600	645	2643
120832	600	601	645	2655
121856	601	600	644	2664
122880	601	601	645	2682
123904	600	600	645	2700
124928	601	600	644	2715
125952	600	600	644	2722
126976	600	600	644	2739
128000	600	600	645	2755
129024	600	600	645	2760
130048	600	600	645	2776
131072	600	600	645	2785
131072	600	600	644	2789
139264	600	600	644	2871
147456	600	600	644	2954
155648	600	600	644	3022
163840	601	600	644	3083
172032	600	600	644	3144
180224	600	601	644	3197
188416	600	601	644	3248
196608	601	600	644	3290
204800	600	600	645	3326
212992	601	601	644	3366
221184	600	601	644	3399
229376	600	600	644	3429
237568	601	600	645	3464
245760	600	600	644	3490
253952	600	601	644	3518
262144	600	600	644	3542
270336	600	600	644	3567
278528	601	600	644	3589
286720	601	601	644	3604

294912	601	600	644	3625
303104	601	601	644	3704
311296	600	600	644	3657
319488	601	600	644	3672
327680	600	600	644	3691
335872	601	601	644	3705
344064	601	600	644	3726
352256	600	601	644	3731
360448	600	601	645	3751
368640	600	601	645	3821
376832	601	601	644	3941
385024	601	605	645	3941
393216	600	600	645	3835
401408	601	601	644	3821
409600	601	601	645	3961
417792	600	601	645	3824
425984	601	601	644	4177
434176	601	601	646	4531
442368	601	601	644	4917
450560	601	601	646	3889
458752	601	601	645	4049
466944	601	601	645	4158
475136	601	601	645	4029
483328	601	601	646	4105
491520	601	601	646	3996
499712	601	601	646	4589
507904	601	601	645	-
516096	601	601	645	-
524288	601	601	647	-
532480	601	602	647	-
540672	601	602	646	-
548864	602	602	646	-

557056	602	602	646	-
565248	602	602	645	-
573440	602	603	647	-
581632	603	603	646	-
589824	603	602	648	-
598016	603	601	647	-
606208	603	602	647	-
614400	602	602	647	-
622592	603	604	647	-
630784	603	603	647	-
638976	603	603	650	-
647168	603	603	647	-
655360	603	603	650	-
663552	604	603	653	-
671744	603	603	652	-
679936	603	604	649	-
688128	604	603	648	-
696320	604	603	649	-
704512	603	604	650	-
712704	604	603	649	-
720896	604	604	650	-
729088	604	604	654	-
737280	604	603	651	-
745472	605	604	651	-
753664	604	607	650	-
761856	605	604	652	-
770048	604	604	653	-
778240	606	605	653	-
786432	606	605	653	-
794624	605	605	653	-
802816	606	605	652	-
811008	606	605	652	-

819200	606	607	652	-
827392	606	605	652	-
835584	606	605	652	-
843776	605	605	652	-
851968	605	606	652	-
860160	606	606	655	-
868352	607	607	655	-
876544	607	606	654	-
884736	606	606	656	-
892928	607	607	655	-
901120	606	607	658	-
909312	607	610	658	-
917504	607	607	655	-
925696	608	607	654	-
933888	607	608	657	-
942080	607	606	656	-
950272	608	608	655	-
958464	607	610	655	-
966656	607	611	658	-
974848	608	613	657	-
983040	609	608	656	-
991232	608	609	658	-
999424	608	609	657	-
1007616	609	609	658	-
1015808	609	610	659	-
1024000	609	610	657	-
1032192	610	610	660	-
1040384	609	610	661	-
1048576	609	610	658	-
1056768	610	610	659	-
1064960	609	611	660	-
1073152	609	611	663	-

1081344	610	609	662	-
1089536	610	611	661	-
1097728	609	609	662	-
1105920	610	608	662	-
1114112	609	610	662	-
1122304	609	610	661	-
1130496	610	610	661	-
1138688	609	610	664	-
1146880	611	612	667	-
1155072	611	612	664	-
1163264	612	610	666	-
1171456	612	613	673	-
1179648	611	613	666	-
1187840	611	613	666	-
1196032	611	613	666	-
1204224	610	613	670	-
1212416	610	613	670	-
1220608	613	611	671	-
1228800	610	610	672	-
1236992	611	611	669	-
1245184	611	610	673	-
1253376	614	610	672	-
1261568	612	610	672	-
1269760	611	611	674	-
1277952	611	612	676	-
1286144	611	611	675	-
1294336	614	612	675	-
1302528	612	612	675	-
1310720	608	609	687	-
1318912	613	610	690	-
1327104	612	611	666	-
1335296	611	611	663	-

1343488	611	612	665	-
1351680	611	611	665	-
1359872	611	611	665	-
1368064	614	612	665	-
1376256	613	612	664	-
1384448	612	613	664	-
1392640	612	613	688	-
1400832	610	612	665	-
1409024	613	612	698	-
1417216	613	612	674	-
1425408	614	612	686	-
1433600	614	613	684	-
1441792	613	612	663	-
1449984	611	613	682	-
1458176	616	613	689	-
1466368	612	614	670	-
1474560	612	613	670	-
1482752	613	613	664	-
1490944	612	613	684	-
1499136	612	612	665	-
1507328	614	613	666	-
1515520	613	614	665	-
1523712	612	615	679	-
1531904	612	613	665	-
1540096	613	614	704	-
1548288	613	613	680	-
1556480	613	612	680	-
1564672	612	613	667	-
1572864	616	614	667	-
1581056	615	614	681	-
1589248	612	615	667	-
1597440	613	612	666	-

1605632	614	613	665	-
1613824	615	614	667	-
1622016	613	614	677	-
1630208	613	612	668	-
1638400	616	613	697	-
1646592	615	615	672	-
1654784	615	613	670	-
1662976	616	614	670	-
1671168	615	614	668	-
1679360	615	615	686	-
1687552	614	614	668	-
1695744	615	614	671	-
1703936	614	614	669	-
1712128	614	614	667	-
1720320	614	615	672	-
1728512	616	616	668	-
1736704	616	616	670	-
1744896	615	614	682	-
1753088	614	615	682	-
1761280	614	615	676	-
1769472	614	615	676	-
1777664	615	616	671	-
1785856	618	615	667	-
1794048	617	616	667	-
1802240	614	616	678	-
1810432	615	614	661	-
1818624	616	617	659	-
1826816	616	616	692	-
1835008	616	616	673	-
1843200	616	616	677	-
1851392	616	616	673	-
1859584	617	616	679	-

1867776	615	615	675	-
1875968	617	615	675	-
1884160	618	619	665	-
1892352	615	618	668	-
1900544	618	618	681	-
1908736	618	619	679	-
1916928	617	620	682	-
1925120	617	621	666	-
1933312	617	619	685	-
1941504	616	616	668	-
1949696	613	613	683	-
1957888	611	611	666	-
1966080	611	608	670	-
1974272	612	607	668	-
1982464	613	611	684	-
1990656	613	612	668	-
1998848	614	613	685	-
2007040	614	614	670	-
2015232	614	614	683	-
2023424	614	615	687	-
2031616	615	615	695	-
2039808	615	614	681	-
2048000	615	614	698	-
2056192	615	615	690	-
2064384	615	615	702	-
2072576	615	615	686	-
2080768	615	615	679	-
2088960	616	616	693	-