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

«Влияние кэш-памяти на время обработки массивов»

студента 2 курса, группы 21204

Осипова Александра Александровича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
Кандидат технических наук
Власенко А. Ю.

Новосибирск 2022

СОДЕРЖАНИЕ

ЦЕЛЬ

1. Исследование зависимости времени доступа к данным в памяти от их объема.
2. Исследование зависимости времени доступа к данным в памяти от порядка их обхода.

ЗАДАНИЕ

1. Написать программу, многократно выполняющую обход массива заданного размера тремя способами (прямой, обратный и случайный).
2. Для каждого размера массива и способа обхода измерить среднее время доступа к одному элементу (в тактах процессора). Построить графики зависимости среднего времени доступа от размера массива. Каждый последующий размер массива отличается от предыдущего не более, чем в 1.2 раза.
3. Определить размеры кэш-памяти точным образом.
4. На основе анализа полученных графиков:
    - оценить размеры кэш-памяти различных уровней, обосновать ответ, сопоставить результат с известными реальными значениями;
    - определить размеры массива, при которых время доступа к элементу массива при случайном обходе больше, чем при прямом или обратном; объяснить причины этой разницы во временах.
5. Составить отчет по лабораторной работе.

ОПИСАНИЕ РАБОТЫ

Программа обхода массива тремя способами была реализована на сервере кафедры.

Результаты измерений были записаны в csv таблицу.

Описание констант:

- N_MIN = 256 // 1 Кбайт = 1024 байт = 256 элементов типа int
- N_MAX = 8388608 // 32 Мбайт = 32 * 1024 * 1024 байт = 8388608 элементов типа int
- STEP = 1.2

Функция int main(), отражающая логику всей программы:

```
int main() {
	std::ofstream output;
	output.open("output.csv");
	output << "N;DIRECT;BACK;RANDOM;\n";

	int N = N_MIN;
	while (N <= N_MAX) {
		output << N * sizeof(int) << ";";

		int* arr = GetDirectElem(N);
		output << GetTestResults(arr, N);
		output << ";";

		arr = GetBackElem(N);
		output << GetTestResults(arr, N);
		output << ";";

		arr = GetRandElem(N);
		output << GetTestResults(arr, N);
		output << ";\n";

		N *= STEP;
	}
	output.close();
	return 0;
}
```

На каждой итерации цикла в output.csv записываются результаты измерений среднего времени доступа к элементу (в тактах процессора) при разном обходе массива размера N. Параметр N, начиная со значения N_MIN, увеличивается с каждой итерации в 1.2 раза, пока не станет больше значения N_MAX.

Алгоритмы, генерирующие массивы, и их реализация

1. Алгоритм построения массива для прямого обхода.

Создаем массив на size элементов. Последнему элементу присваиваем значение 0, так как попав в него, мы должны взять его значение и вернуться к первому элементу. Каждому оставшемуся элементу под номером i присваиваем значение i+1.

Реализация:

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

2. Алгоритм построения массива для обратного обхода.

Создаем массив на size элементов. Первому элементу присваиваем значение size-1, так как попав в него, мы должны перейти по его значению и вернуться в конец массива. Каждому оставшемуся элементу под номером i присваиваем номер i-1.

Реализация:

```
int* GetBackElem(int size) {
	int* arr = new int[size];
	arr[0] = size - 1;
	for (int i = size - 1; i > 0; --i) {
		arr[i] = i - 1;
	}
	return arr;
}
```

3. Алгоритм построения массива для случайного обхода.

Создаем массив на size элементов и инициализируем его значением -1, так как мы ни один элемент еще не посетили. Далее создаем цикл на size-1 итераций – столько элементов нам нужно случайным образом означить. Начинаем с первого элемента массива (cur = 0). На каждой итерации ищем значение для arr[cur] по следующему принципу: выбираем случайное число от 0 до size-1 до тех пор, пока оно не станет отличаться от индекса текущего элемента, и это число мы бы не использовали ранее. Найдя такое число next, мы присваиваем его элементу под номер cur и далее проделываем тоже самое, но только для элемента arr[next]. После того, как мы проинициализировали size -1 элемент, последнему оставшемуся мы присвоим значение 0. Так как он последний, то перейдя по его значению, мы вернемся в начало массива, тем самым получив цикл.

Реализация:

```
int FindNext(int cur, int* arr, int size) {
	int next = 0;
	do {
		next = rand() % size;
	} while ((arr[next] != -1) || (next == cur));
	return next;
}
```

```
int* GetRandElem(int size) {
	int* arr = new int[size];
	for (int i = 0; i < size; ++i) {
		arr[i] = -1;
	}
	srand(time(NULL));
	int cur = 0, next = 0;
	for (int i = 0; i < size - 1; ++i) {
		next = FindNext(cur, arr, size);
		arr[cur] = next;
		cur = next;
	}
	arr[cur] = 0;

	return arr;
}
```

Обходы массивов, измерения времени

Для чистоты эксперимента перед каждым счетом и измерением времени выполняется прогрев процессора, чтобы вывести процессор на фиксированную частоту, и прогрев кэша, чтобы вынести оттуда «мусорные» данные. Чтобы прогреть процессор, выполняется умножение двух матриц размером 100 x 100. Для прогревания кэша выполняется предварительный прогон цикла. Так как программа будет компилироваться на уровне оптимизации O1, нужно продемонстрировать компилятору результат работы цикла, чтобы он не удалил его в качестве оптимизации.

```
void WarmCPU() {
	std::ifstream input;
	input.open("matrix.txt");
	Matrix A(100);
	Matrix B(100);
	A.init(input);
	B.init(input);

	Matrix C = A * B;
}
```

```
void WarmCache(int* arr, int size) {
	int k = 0;
	for (int i = 0; i < size * K; ++i) {
		k = arr[k];
	}
	if (k == 12345) std::cout << "Wow!";
}
```

Для измерения времени в тактах процессора была написана функция uint64_t GetTSC().

```
uint64_t GetTSC() {
	uint64_t highPart, lowPart;
	asm volatile("rdtsc\n":"=a"(lowPart), "=d"(highPart));
	return (highPart << 32) | (lowPart);
}
```

Результат ассемблерной вставки:

```
    rdtsc

    movq    %rax, -8(%rbp)
    movq    %rdx, -16(%rbp)
    movq    -16(%rbp), %rax
    salq    $32, %rax
    orq     -8(%rbp), %rax
```

Функция, возвращающая результат измерения времени обхода массива:

```
uint64_t GetTestResults(int* arr, int size) {
	uint64_t start, end;
	uint64_t accessTime = 0, minAccessTime = INT_MAX;

	WarmCPU();
	WarmCache(arr, size);

	for (int count = 0; count < ATTEMPTS; ++count) {
		int k = 0;
		start = GetTSC();
		for (int i = 0; i < size * K; ++i) {
			k = arr[k];
		}
		end = GetTSC();
		if (k == 12345) std::cout << "Wow!";
		accessTime = (end - start) / (size * K);
		if (accessTime < minAccessTime) minAccessTime = accessTime;
	}
	delete[] arr;
	return minAccessTime;
}
```

Для большей объективности цикл с обходом массива прогоняется 3 раза и выбирается наименьшее время доступа к элементу.

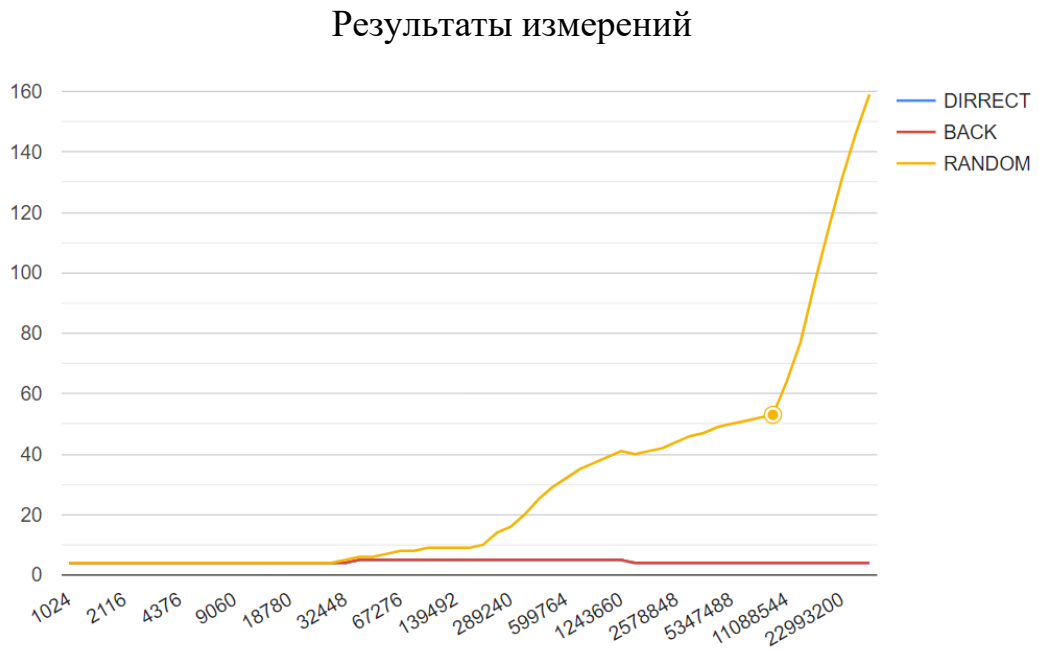

График результатов измерений

На графике видно, что в случае прямого и обратного обхода среднее время доступа к элементу массива не меняется в зависимости от его размера. Это связано с тем, что для процессора такой вид обхода является предсказуемым, поэтому он заранее подгружает в кэш первого уровня следующие элементы массива. При случайном обходе массива процессор не может распознать никакую закономерность, поэтому вынужден брать элементы из первого, второго или третьего уровня кэша, если массив помещается в кэш-памяти. При больших размерах процессор должен брать элементы массивы из оперативной памяти, что еще сильнее сказывается на времени доступа к элементам.
Размер L1: ~32 Кбайт
Размер L2: ~250 Кбайт
Размер L3: ~9 Мбайт

Данные, полученные командой lscpu

```
Ядер на сокет:                   6
Сокетов:                         2
```

```
L1d cache:                       384 KiB
L1i cache:                       384 KiB
L2 cache:                        3 MiB
L3 cache:                        24 MiB
```

Заметим, что значения размеров, показанные на графике, почти совпадают с фактическими размерами уровней кэшей. В случае L1 и L2 число нужно поделить на 12 (2 процессора, на каждом по 6 ядер), а в случае L3 – на 2 (третий уровень кэша общий для всех ядер каждого процессора). Получим 32 Кбайт, 256 Кбайт и 12 Мбайт соответственно.

ЗАКЛЮЧЕНИЕ

В рамках данной лабораторной работы была исследована зависимость времени доступа к данным в памяти от их объема и порядка обхода. Так, в случае прямого и обратного обходов, время доступа не зависит от объема данных, так как процессор, обнаруживая это, подгружает следующие элементы массива в кэш первого уровня, из-за чего среднее время остается без изменений. В случае же случайного обхода, процессор не может оптимизировать чтение данных, так как не видит закономерности в их расположении, поэтому вынужден читать элементы оттуда, где они непосредственно лежат. В зависимости от размера массива, элементы могут лежать в разных уровнях кэша и даже в памяти, а расположение элемента сильно влияет на время доступа к нему.

Приложение 1. Листинг программы

```
#include <cstdlib>
#include <iostream>
#include <stdint.h>
#include <limits.h>
#include <fstream>
#include "matrix.h"

#define N_MIN 256
#define N_MAX 8388608
#define K 5
#define STEP 1.2
#define ATTEMPTS 5

int* GetDirectElem(int size) {
	int* arr = new int[size];
	arr[size - 1] = 0;
	for (int i = 0; i < size - 1; ++i) {
		arr[i] = i + 1;
	}
	return arr;
}

int* GetBackElem(int size) {
	int* arr = new int[size];
	arr[0] = size - 1;
	for (int i = size - 1; i > 0; --i) {
		arr[i] = i - 1;
	}
	return arr;
}

int FindNext(int cur, int* arr, int size) {
	int next = 0;
	do {
		next = rand() % size;
	} while ((arr[next] != -1) || (next == cur));
	return next;
}

int* GetRandElem(int size) {
	int* arr = new int[size];
	for (int i = 0; i < size; ++i) {
		arr[i] = -1;
	}
	srand(time(NULL));
	int cur = 0, next = 0;
	for (int i = 0; i < size - 1; ++i) {
		next = FindNext(cur, arr, size);
		arr[cur] = next;
		cur = next;
	}
	arr[cur] = 0;
	return arr;
}

void WarmCPU() {
	std::ifstream input;
	input.open("matrix.txt");
	Matrix A(100);
	Matrix B(100);
	A.init(input);
	B.init(input);
```

```
	Matrix C = A * B;
}

void WarmCache(int* arr, int size) {
	int k = 0;
	for (int i = 0; i < size * K; ++i) {
		k = arr[k];
	}
	if (k == 12345) std::cout << "Wow!";
}

uint64_t GetTSC() {
	uint64_t highPart, lowPart;
	asm volatile("rdtsc\n":"=a"(lowPart), "=d"(highPart));
	return (highPart << 32) | (lowPart);
}

uint64_t GetTestResults(int* arr, int size) {
	uint64_t start, end;
	uint64_t accessTime = 0, minAccessTime = INT_MAX;

	WarmCPU();
	WarmCache(arr, size);

	for (int count = 0; count < ATTEMPTS; ++count) {
		int k = 0;
		start = GetTSC();
		for (int i = 0; i < size * K; ++i) {
			k = arr[k];
		}
		end = GetTSC();
		if (k == 12345) std::cout << "Wow!";
		accessTime = (end - start) / (size * K);
		if (accessTime < minAccessTime) minAccessTime = accessTime;
	}
	delete[] arr;
	return minAccessTime;
}

int main() {
	std::ofstream output;
	output.open("output.csv");
	output << "N;DIRECT;BACK;RANDOM;\n";

	int N = N_MIN;
	while (N <= N_MAX) {
		output << N * sizeof(int) << ";";

		int* arr = GetDirectElem(N);
		output << GetTestResults(arr, N) << ";";

		arr = GetBackElem(N);
		output << GetTestResults(arr, N) << ";";

		arr = GetRandElem(N);
		output << GetTestResults(arr, N) << ";\n";

		N *= STEP;
	}
	output.close();
	return 0;
}
```

Приложение 2. Таблица с результатами измерений

N	DIRECT	BACK	RANDOM
1024	4	4	4
1228	4	4	4
1472	4	4	4
1764	4	4	4
2116	4	4	4
2536	4	4	4
3040	4	4	4
3648	4	4	4
4376	4	4	4
5248	4	4	4
6296	4	4	4
7552	4	4	4
9060	4	4	4
10872	4	4	4
13044	4	4	4
15652	4	4	4
18780	4	4	4
22536	4	4	4
27040	4	4	4
32448	4	4	5
38936	5	5	6
46720	5	5	6
56064	5	5	7
67276	5	5	8
80728	5	5	8
96872	5	5	9
116244	5	5	9
139492	5	5	9
167388	5	5	9
200864	5	5	10
241036	5	5	14
289240	5	5	16
347088	5	5	20
416504	5	5	25
499804	5	5	29
599764	5	5	32
719716	5	5	35
863656	5	5	37
1036384	5	5	39
1243660	5	5	41
1492392	4	4	40
1790868	4	4	41
2149040	4	4	42
2578848	4	4	44
3094616	4	4	46

3713536	4	4	47
4456240	4	4	49
5347488	4	4	50
6416984	4	4	51
7700380	4	4	52
9240456	4	4	53
11088544	4	4	64
13306252	4	4	77
15967500	4	4	96
19161000	4	4	114
22993200	4	4	131
27591840	4	4	146
33110208	4	4	159