

**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
Практическая работа №8**

ВЛИЯНИЕ КЭШ-ПАМЯТИ НА ВРЕМЯ ОБРАБОТКИ МАССИВОВ
студента 2 курса, группы 23201

Сорокина Матвея Павловича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
А.С. Матвеев

Новосибирск 2024

СОДЕРЖАНИЕ

ЦЕЛЬ.....	3
ЗАДАНИЕ	3
ОПИСАНИЕ РАБОТЫ	4
ЗАКЛЮЧЕНИЕ	7
ПРИЛОЖЕНИЯ.....	8
Приложение 1: Исходный код программы main2.cpp	8
Приложение 2: пример результата работы программы.....	11
Приложение 3: листинг bash-скрипта для создания графика с использованием Gnuplot и результатов работы main2.cpp, записанных в log.txt	11

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.
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.
3. На основе анализа полученных графиков:
 - определить 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.
4. Составить отчет по лабораторной работе.

ОПИСАНИЕ РАБОТЫ

Кэш-память процессора (CPU Cache Memory) — это тип временного хранилища данных, расположенного непосредственно на процессоре. Она используется для повышения эффективности обработки данных, так как хранит небольшие, часто запрашиваемые фрагменты данных, готовые к быстрому доступу. Кэш-память делится на несколько уровней: L1, L2, L3. Эти уровни различаются по расположению, скорости и объему памяти. Кэш-память значительно быстрее RAM, зачастую в 10-100 раз, и физически располагается очень близко к ядрам процессора.

Причина, по которой кэш-память *SRAM* (статическая оперативная память) не используется вместо основной оперативной памяти компьютера *DRAM* (динамическая оперативная память), связана с её стоимостью. Объем кэш-памяти на процессоре относительно невелик, измеряется в килобайтах или мегабайтах, а не в гигабайтах, так как изготовление таких больших объемов *SRAM* было бы чрезмерно дорогим.

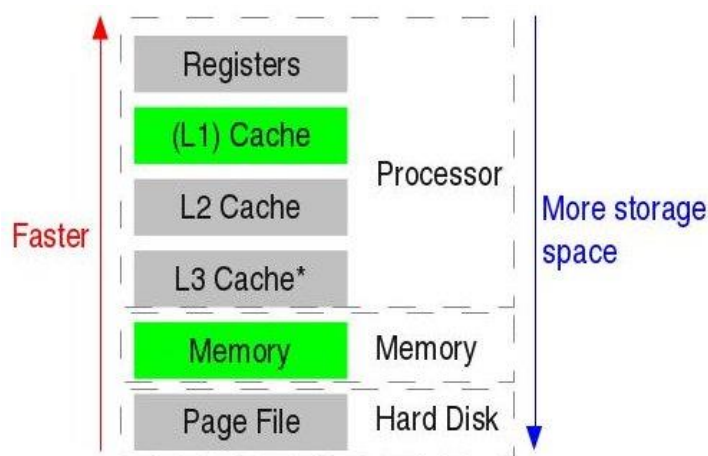


Рис. 1. Иерархия памяти

Уровни кэш памяти процессора *13th Gen Intel(R) Core(TM) i7-13700H*:

- **L1d (Data Cache L1):**
Размер: 544 kB (14 ядер):
Performance cores (6):
Data: $288\text{kB} = 6 * 48\text{ kB}$
Efficient cores (8):
Data: $256\text{kB} = 8 * 32\text{ kB}$
- **L1i (Instructions Cache L1):**
Размер: 704 kB (14 ядер):
Performance cores (6):

Instructions: $192 \text{ kB} = 6 * 32 \text{ kB}$

Efficient cores (8):

Instructions: $512 \text{ kB} = 8 * 64 \text{ kB}$

- **L2 Cache (Medium Level Cache):**

Размер: $11.5 \text{ MB} = 11776 \text{ kB}$ (8 ядер)

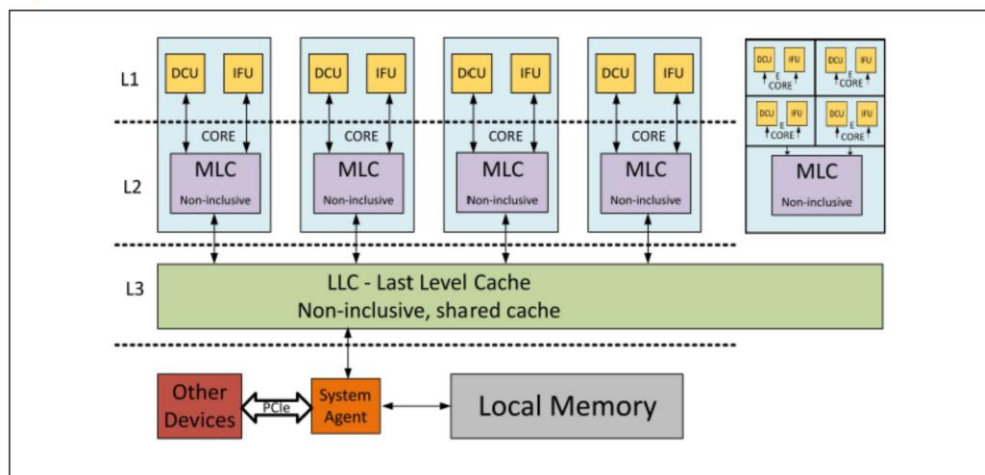
Performance cores (6): $7680 \text{ kB} = 6 * 1280 \text{ kB}$

Efficient cores (2): $4 \text{ MB} = 2 * 2048 \text{ kB}$

- **L3 Cache (Last Level Cache):**

Размер: 24 MB

Hybrid Cache



NOTES

1. L1 Data cache (DCU) - 48KB (P-core) - 32KB (E-Core)
2. L1 Instruction cache (IFU) - 32KB (P-Core) - 64KB (E-Core)
3. MLC - Mid Level Cache - 1.25MB (P-Core) - 2MB (shared by 4 E-Cores)

Рис 2. Устройство процессоров 13 и 14 поколения Intel

Итого получаем примерные точки, когда массивы начинают выходить за пределы кэшей:

- **До 48 kB:** данные вмещаются в L1d, самый быстрый уровень.
- **48 kB - 1280 kB:** данные начинают заполнять L2.
- **1280 kB - 24 MB:** данные заполняют L3.
- **Больше 24 MB:** массив полностью выходит за пределы кэшей.

Описание обходов элементов массива:

1. **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 обходим массив таким образом несколько раз.
2. **Обратный обход** – прямой обход, но с условием, что теперь обход идёт с конца массива линейным порядком.
3. **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.

Оценка размера кэша программой с помощью создания графиков зависимости среднего времени чтения элемента массива от размера массива при различных способах обхода реализована (см. Приложение 1, 2).

ЗАКЛЮЧЕНИЕ

В ходе данной работы были установлены размеры кэшей процессора ***13th Gen Intel(R) Core(TM) i7-13700H*** и выяснено, насколько существенен прирост времени обращения при "переходе" с одного кэша на другой.

ПРИЛОЖЕНИЯ

Приложение 1: Исходный код программы *main2.cpp*

```
#include <fstream>
#include <ctime>
#include <cmath> // INFINITY
#include <algorithm> // std::swap
#include <chrono>
#include <iostream>
#include <numeric> // std::iota

#define STEP (1 * 1024) // шаг увеличения размера массива
#define MAX_SIZE (3 * 1024 * 1024) // максимальный размер массива

#define RUNS 4 // количество запусков

// перемешивание элементов массива СЛУЧАЙНО
void Shuffle(size_t* array, size_t n) {
    if (n < 2) { return; } // массив слишком мал для перемешивания
    for (size_t i = 0; i < n - 1; i++) {
        size_t j = i + rand() / (RAND_MAX / (n - i) + 1);
        // swap(&array[i], &array[j]);
        std::swap(array[i], array[j]);
    }
}

// Поиск значения в массиве
size_t Find(size_t* array, size_t n, size_t value) {
    for (size_t i = 0; i < n; i++) {
        if (array[i] == value) {
            return i;
        }
    }
}

return -1;
}

// С атрибутом inline компилятор не будет встраивать эту функцию в место вызова,
// а будет всегда выполнять фактический вызов функции.
// Для отладки
```



```

__attribute__((noinline)) uint64_t ff(size_t data[], size_t size) {
    uint64_t sum = 0;
    size_t prev = 0;
    // основной цикл, многократно проходящий по данным и суммирующий индексы
    for (size_t i = 0; i < size; i++) {
        size_t current = data[prev];
        sum += current;
        prev = current;
    }
    return sum;
}

// функция для тестирования времени доступа к данным
long f(size_t size) {
    // выделение памяти для индексов и заполнение их значениями от 0 до size-1
    size_t* indices = new size_t[size];
    std::iota(indices, indices + size, 0);
    Shuffle(indices, size); // перемешка индексов

    // Находим индекс, равный 0, и перемещаем его в конец массива
    size_t zero = Find(indices, size, 0);
    std::swap(indices[zero], indices[size - 1]);

    // массив данных, связывая элементы с индикаторами
    size_t* data = new size_t[size];
    size_t index = 0;
    for (size_t i = 0; i < size; i++) {
        data[index] = indices[i];
        index = indices[i];
    }
    delete[] indices;

    const auto start = std::chrono::high_resolution_clock::now();
    uint64_t sum = ff(data, size);
    const auto end = std::chrono::high_resolution_clock::now();
    delete[] data;

    std::fstream dev_null("/dev/null", std::ios::out);
    dev_null << sum << std::endl;

    const std::chrono::duration<double, std::nano> elapsed_time = end - start;
}

```

```

// время выполнения в наносекундах
return elapsed_time.count();
}

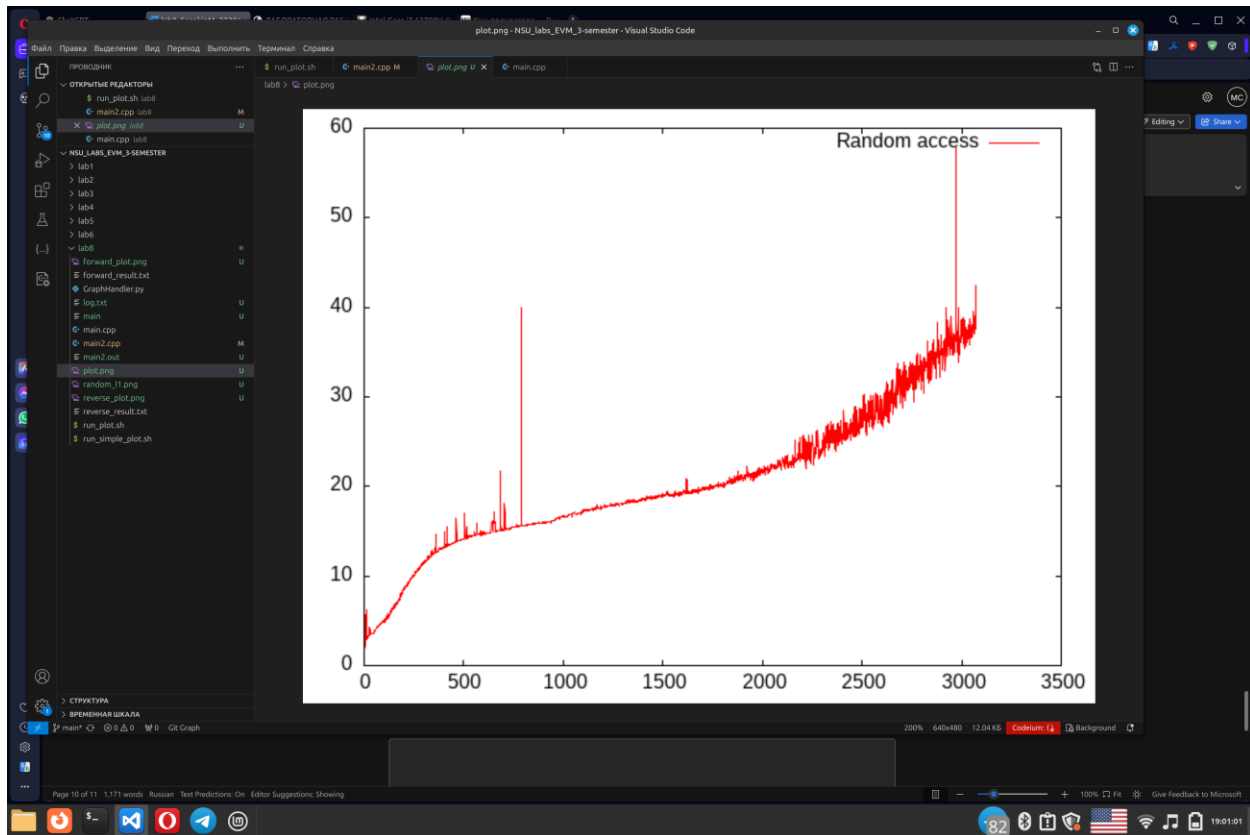
int main() {
for (size_t i = STEP; i < MAX_SIZE; i += STEP) {
double kib = i / 1024.0; // Перевод размера в Kib (был в байтах)

double min_time = INFINITY;
for (size_t j = 0; j < RUNS; j++) {
double f_time = f(i); // Время выполнения функции f
double curr_time = f_time / i;
min_time = std::min(min_time, curr_time);
}
// Вывод размера массива в KiB и минимального времени
// printf("%f\t%f\n", kib, min_time);
std::cout << kib << "\t" << min_time << std::endl;
}

return 0;
}

```

Приложение 2: пример результата работы программы main2.cpp



Приложение 3: листинг bash-скрипта для создания графика с использованием *Gnuplot* и результатов работы main2.cpp, записанных в *log.txt*

```
#!/bin/bash

g++ -O1 main2.cpp -o main2.out
time(./main2.out > log.txt)

gnuplot <<EOF
set terminal png
set output 'plot.png'
plot 'log.txt' with lines linecolor rgb "red" title 'Random access'
set output
EOF

echo "Plot saved as plot.png"
```