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

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.

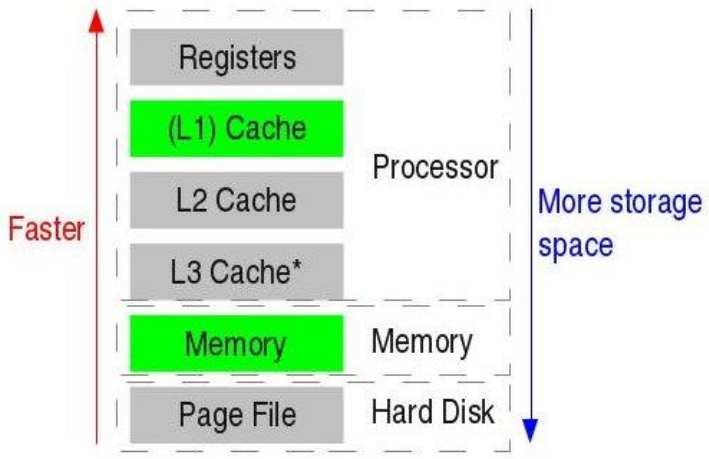


Рис. 1. Иерархия памяти

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: 288kB = 6 * 48 kB
  Efficient cores (8):
  Data: 256kB = 8 * 32 kB

- **L1i (Instructions Cache L1):**
  Размер: 704 kB (14 ядер):
  Performance cores (6):

Instructions: 192 kB = 6 * 32 kB
Efficient cores (8):
Instructions: 512 kB = 8 * 64 kB

- **L2 Cache (Medium Level Cache):**
  Размер: 11.5 MB = 11776 kB (8 ядер)
  Performance cores (6): 7680 kB = 6 * 1280 kB
  Efficient cores (2): 4 MB = 2 * 2048 kB

- **L3 Cache (Last Level Cache):**
  Размер: 24 MB

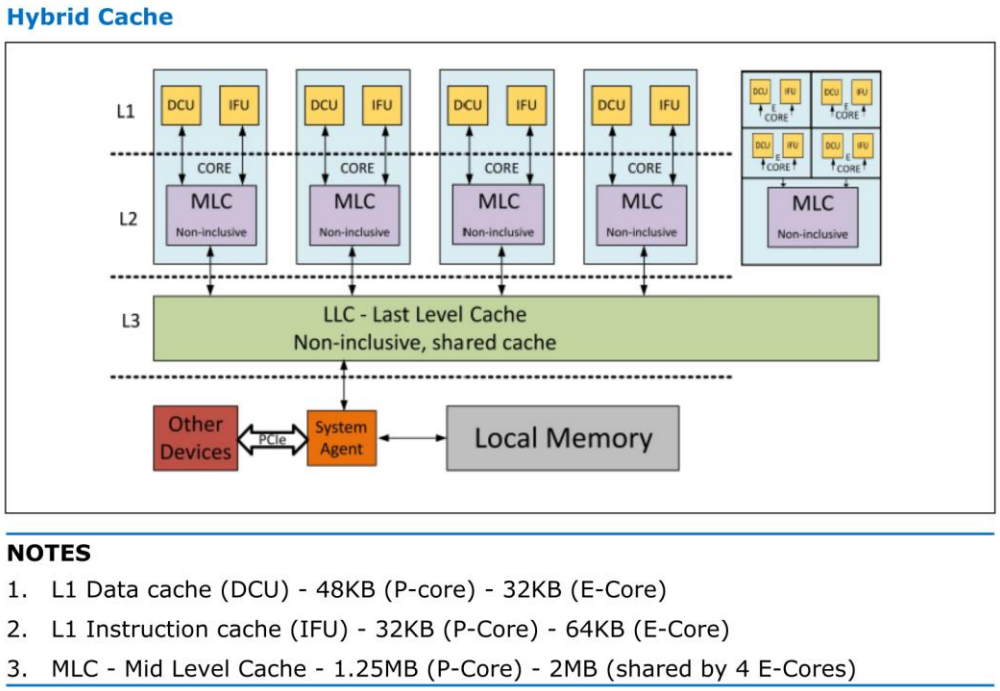


Рис 2. Устройство процессоров 13 и 14 поколения Intel

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 для вывода среднего времени чтения элемента массива при прямом и обратном обходе ***main.cpp***

```
#include <iostream>
#include <vector>
#include <chrono>
#include <random>
#include <algorithm>
#include <numeric>
#include <iomanip>
#include <fstream>

unsigned long long MeasureAccessTime(const std::vector<int>& array, std::fstream& dev_null) {
auto start = std::chrono::high_resolution_clock::now(); // Начало замера

size_t sum = 0;
size_t prev = 0;
for (size_t i = 0; i < array.size(); i++) {
sum += array[prev]; // Доступ к элементу по указанному индексу
prev = array[prev];
}

auto end = std::chrono::high_resolution_clock::now(); // Конец замера
std::chrono::duration<double, std::nano> elapsed_time = end - start; // Разница во времени в наносекундах

dev_null << sum << std::endl;
return static_cast<unsigned long long>(elapsed_time.count()); // Возвращаем время в наносекундах
}

std::vector<size_t> GenerateIndices(size_t size, const std::string& mode) {
std::vector<size_t> indices(size);
std::iota(indices.begin(), indices.end(), 0); // Заполняем индексами 0, 1, 2 и т.д.

if (mode == "reverse") {
std::reverse(indices.begin(), indices.end()); // Обратный порядок
}
```

```
return indices;
}

int main() {
std::vector<size_t> sizes;
for (size_t i = 1*1024; i <= 2 * 1024*1024; i += 1024) {
sizes.push_back(i);
}

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

std::fstream forward_res("forward_result.txt", std::ios::out);
std::fstream reverse_res("reverse_result.txt", std::ios::out);

std::string modes[] = {"forward", "reverse"};

for (size_t size : sizes) {
std::vector<int> array(size * sizeof(int)); // создаем массив нужного размера в байтах

for (const std::string& mode : modes) {
const int repetitions = 10; // Количество повторений для замера
auto indices = GenerateIndices(size * sizeof(int), mode);

size_t index = 0;
for (size_t i = 0; i < size * sizeof(int); ++i) {
array[index] = indices[i];
index = indices[i];
}

unsigned long long min_time_ns = 1000000000;
for (int r = 0; r < repetitions; r++) {
min_time_ns = std::min(min_time_ns, MeasureAccessTime(array, dev_null));
}

double avg_time_per_element_ns = min_time_ns / size;

// size * sizeof(int) - размер массива в байтах
// avg_time_per_element_ns - среднее время доступа к одному элементу в наносекундах
if (mode == "forward") {
forward_res << size / 1024 << " " << avg_time_per_element_ns << std::endl;
} else if (mode == "reverse") {
```

```
        reverse_res << size / 1024 << " " << avg_time_per_element_ns << std::endl;
        }
        }
        }

        return 0;
        }
```

Приложение 2: Исходный код программы для вывода среднего времени чтения элемента массива при случайном обходе ***main2.cpp***

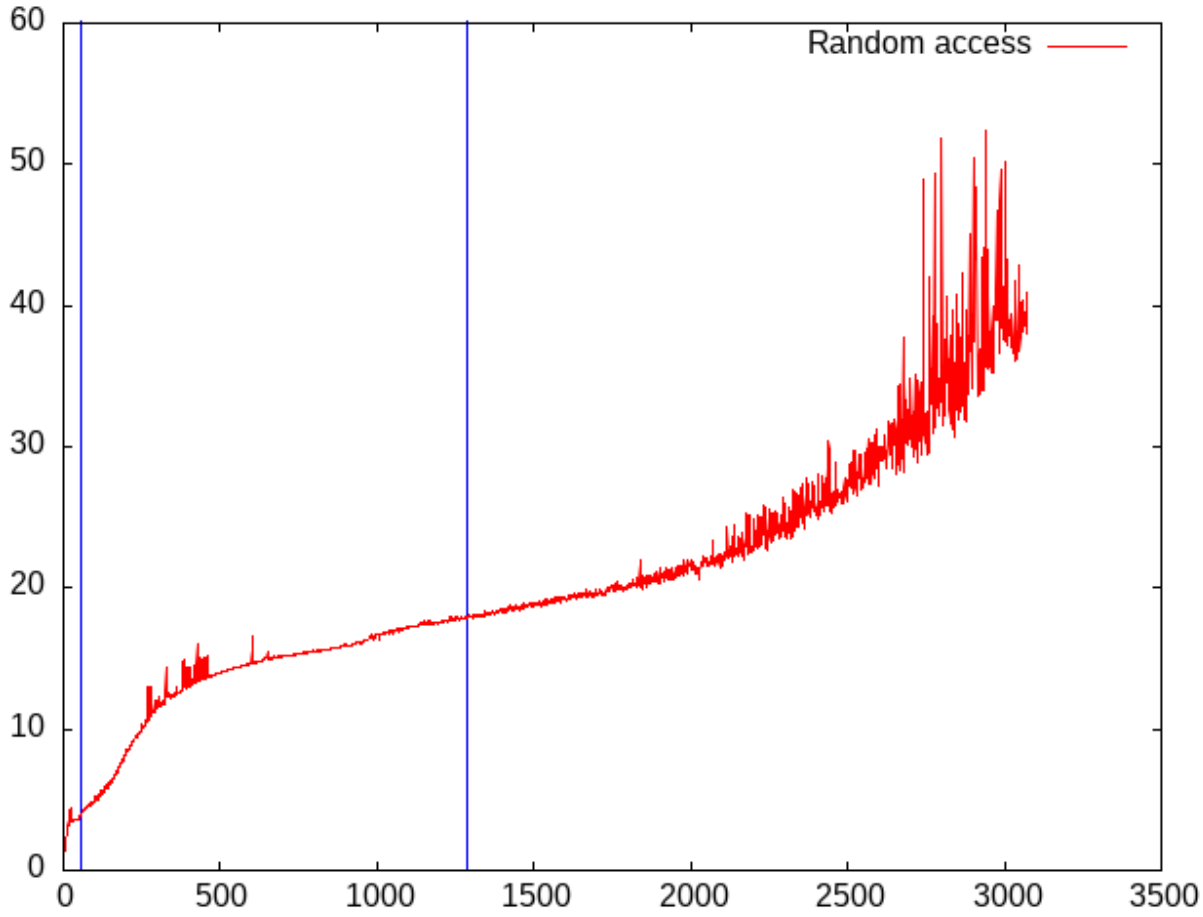
```
        #include <fstream>
        #include <ctime>
        #include <cmath> // INFINITY
        #include <algorithm> // std::swap
        #include <chrono>
        #include <iostream>
        #include <numeric> // std::iota
        #include <random>

        #include <cstdlib> // rand

        #define STEP (1 * 1024) // шаг увеличения размера массива
        #define MAX_SIZE (2 * 1024 * 1024) // максимальный размер массива

        // #define LOOPS 3 // количество повторений
        #define RUNS 8 // количество запусков

        // перемешивание элементов массива СЛУЧАЙНО
        // void Shuffle(size_t* array, size_t n) {
        // if (n < 2) { return; }
        // std::random_device rd; // возвращает случайное значение
        // std::mt19937 gen(rd()); // генератор псевдослучайных чисел, реализующий алгоритм Mersenne Twister
        // std::shuffle(array, array + n, gen); // gen используется как генератор случайных чисел
        // }

        void Shuffle(size_t* array, size_t n) {
```

```
        if (n < 2) { return; }
        srand(static_cast<unsigned int>(time(NULL))); // инициализируем генератор rand
случайным значением
        for (size_t i = n - 1; i > 0; --i) {
        size_t j = rand() % (i + 1); // выбираем случайный индекс от 0 до i
        std::swap(array[i], array[j]); // меняем элементы местами
        }
        }

        // Плиск 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 noinline 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) {
        // ыделение памяти для индексов и заполнение их значениями от 0 до size-1
        size_t* indicies = new size_t[size];
        std::iota(indicies, indicies + size, 0);
        Shuffle(indicies, size); // перемешка индексов
```

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

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

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);
// if ((curr_time < min_time) && (curr_time > 0)) {
// min_time = curr_time;
// }
```

```
    }
    // Вывод размера массива в KiB и минимального времени
    std::cout << kib << "\t" << min_time << std::endl;
    }

    return 0;
    }
```

Приложение 3: пример результата работы программы main2.cpp



Приложение 4: листинг bash-скрипта для создания графика с использованием *Gnuplot и* результатов работы main2.cpp, записанных в *log.txt - run_plot.sh*

```
#!/bin/bash
```

```
g++ -O1 main2.cpp -o main2.out
time(./main2.out > random_l2_2.txt)

gnuplot <<EOF
set terminal png
set output 'random_l2_2.png'

set arrow from 48, graph 0 to 48, graph 1 nohead lw 2 lc rgb "blue"
set arrow from 1280, graph 0 to 1280, graph 1 nohead lw 2 lc rgb "blue"

plot 'random_l2_2.txt' with lines linecolor rgb "red" title 'Random access'
EOF

echo "Plot saved as random_l2_2.png"
```