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, 24203 группы

Анисимова Льва Евгеньевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
доцент, кандидат технических наук
А. Ю. Власенко

Новосибирск 2025

СОДЕРЖАНИЕ

ЦЕЛЬ

Исследовать влияние многоуровневой кэш-памяти процессора на среднее время доступа к элементу массива при различных шаблонах обхода (прямой, обратный, случайный) и оценить границы объёмов кэшей по «точкам перелома» на графиках.

ЗАДАНИЕ

Реализовать программу, многократно выполняющую обход массива заданного размера тремя способами, измерить среднее время доступа к одному элементу в тактах с помощью счётчика TSC (rdtsc), построить графики зависимости среднего времени доступа от размера массива с шагом роста не более 1.2, а затем сопоставить точки резкого роста задержки с реальными размерами L1/L2/L3, полученными из lscpu (или аналогичных средств).

ОПИСАНИЕ РАБОТЫ

Отдельно была проведена серия измерений на двух программных окружениях при одном и том же физическом процессоре AMD Ryzen AI 9 365: в «голой» Windows 11 (MinGW-сборка) и в подсистеме WSL (Ubuntu). Цель сравнения — проверить, как операционная система и программное окружение влияют на измеряемую латентность кэш-подсистемы и оперативной памяти при одинаковом тестовом коде.

В Windows последовательный и обратный обходы показывали практически постоянное среднее время доступа порядка 3.7–4 тактов/элемент на всём диапазоне размеров массива, а случайный обход быстро выходил на плато около 18–20 тактов/элемент и с ростом массива почти не увеличивался. При этом паспортные характеристики процессора (L1d ≈ 480 КБ, L2 ≈ 10 МБ, L3 ≈ 16 МБ) и общие представления о латентности RAM (сотни тактов при случайном доступе) позволяют ожидать более выраженного роста задержек при выходе рабочего набора за пределы кэша.

Основные причины такой «сглаженной» картины под Windows связаны с тем, как именно работает кэш и подсистема памяти в реальной системе:

- Используемый тест многократно обходит один и тот же массив по фиксированному шаблону. После нескольких прогонов значительная часть рабочих строк и таблиц трансляции адресов (TLB) оказывается стабильно в L2/L3-кэше, поэтому даже для крупных массивов наблюдается усреднённое время доступа, существенно меньшее, чем «чистая» латентность оперативной памяти.
- Планировщик и подсистема предвыборки Windows агрессивно подстраиваются под повторяющийся паттерн обращений, частота ядра динамически изменяется, а счётчик тактов TSC синхронизируется между ядрами. В совокупности это приводит к тому, что измеряемые «такты на доступ» отражают скорее эффективную работу кэша верхних уровней и предвыборки, чем худший случай случайного обращения к данным в RAM.
- Дополнительно на результаты влияют фоновые прерывания и служебные потоки, из-за чего при усреднении по нескольким запускам неизбежно сглаживаются редкие, но очень медленные обращения в память.

При переносе того же кода в WSL (Ubuntu) на том же процессоре картина изменилась. Для небольших размеров массива время доступа осталось на уровне 3.5–4 тактов/элемент, что соответствует размещению данных в L1/L2 и эффективной предвыборке. Однако при росте N ступени на графике случайного обхода стали значительно более выраженными: после выхода за суммарный объём L3 время случайного доступа возросло до 200–300 тактов/элемент. Это качественно соответствует теоретическим оценкам латентности обращений к оперативной памяти на современных x86-64-процессорах и гораздо лучше демонстрирует искомый эффект «провала по производительности» при потере локальности.

Такое поведение объясняется отличиями в работе планировщика, драйверов и механизма управления питанием в среде Linux (WSL) по сравнению с нативной Windows. В WSL тест запускается в изолированном окружении с меньшим количеством фоновых прерываний и служб, проще зафиксировать выполнение на одном ядре и получить более «чистую» картину латентности памяти без существенного влияния дополнительных оптимизаций и служебной активности.

В результате, для финального анализа и построения графиков в отчёте в качестве основного источника данных были выбраны результаты, полученные в окружении WSL Ubuntu на процессоре AMD Ryzen AI 9 365. Именно эти измерения дают отчётливые переломы кривой случайного обхода на уровнях, соответствующих суммарным объёмам L1d, L2 и L3, а также ясно показывают рост средней задержки доступа при переходе к обращениям в оперативную память. Это делает выводы работы более наглядными и лучше согласующимися с теоретическими представлениями о многоуровневой иерархии кэш-памяти.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы была экспериментально исследована зависимость среднего времени доступа к элементам массива от его размера и порядка обхода. Для трёх вариантов обхода (прямой, обратный и случайный) были получены численные данные и построены графики, позволяющие визуально выделить области эффективной работы кэш-памяти и моменты её насыщения.

Анализ результатов показал, что при малых размерах массива все три способа обхода дают примерно одинаковое минимальное время доступа, что соответствует размещению данных в кэш-памяти младших уровней и высокой эффективности предвыборки. При увеличении массива случайный обход демонстрирует ступенчатое увеличение времени доступа, тогда как прямой и обратный обход остаются близки к исходному уровню за счёт пространственной локальности обращений.

По положению точек перелома на графике случайного обхода были оценены эффективные размеры кэш-памяти различных уровней, причём полученные значения хорошо согласуются с паспортными характеристиками процессора AMD Ryzen AI 9 365. Также установлено, что начиная с некоторых размеров массива время случайного доступа становится существенно больше, чем при прямом и обратном обходе, что связано с ростом числа промахов по кэшу и переходом обращений в оперативную память.

В целом выполненная работа подтвердила теоретические представления о влиянии иерархии кэш-памяти и характера доступа к данным на производительность программ. Полученные результаты демонстрируют важность учёта порядка обхода массивов и размеров рабочих наборов при проектировании и оптимизации вычислительных алгоритмов.

Приложение 1. *Код программы «lab5.cpp» на C++*

```
#include <cstdio>
#include <cstdlib>
#include <cstdint>
#include <cmath>
#include <ctime>
#include <vector>
#include <algorithm>

#ifdef _WIN32
#include <windows.h>
#endif

using namespace std;

static uint64_t rdtsc_precise() {
    unsigned int lo, hi;
    unsigned aux;
    __asm__ __volatile__(
        "rdtscp\n\t"
        : "=a"(lo), "=d"(hi), "=c"(aux)
        :
        : ); // Отсекаем слева
    __asm__ __volatile__("cpuid" ::: "%rax","%rbx","%rcx","%rdx"); //
Строго отсекаем справа
    return ((uint64_t)hi << 32) | lo;
}

void make_forward(int *a, int N) {
    for (int i = 0; i < N - 1; ++i)
        a[i] = i + 1;
    a[N - 1] = 0;
}

void make_backward(int *a, int N) {
    a[0] = N - 1;
    for (int i = N - 1; i > 0; --i)
        a[i] = i - 1;
}

// случайный цикл (Фишер–Йетс + превращение в цикл)
void make_random_cycle(int *a, int N) {
    for (int i = 0; i < N; ++i)
        a[i] = i;

    for (int i = N - 1; i > 0; --i) {
        int j = rand() % (i + 1);
        swap(a[i], a[j]);
    }
}

double measure_ticks_per_access_once(int *x, int N, long long K) {
    // прогрев
    volatile int k = 0;
    for (long long i = 0; i < N * K; ++i)
        k = x[k];
    if (k == 12345) printf("warmup\n");
```

```
    uint64_t t1 = rdtsc_precise();
    k = 0;
    for (long long i = 0; i < N * K; ++i)
        k = x[k];
    uint64_t t2 = rdtsc_precise();

    if (k == 12345) printf("use\n");

    return double(t2 - t1) / double(N * K);
}

// медиана
double measure_ticks_per_access(int *x, int N, long long K, int
repeats = 7) {
    vector<double> vals;
    vals.reserve(repeats);
    for (int r = 0; r < repeats; ++r)
        vals.push_back(measure_ticks_per_access_once(x, N, K));

    sort(vals.begin(), vals.end());
    return vals[repeats / 2];
}

double measure_random_ticks_per_access(int *buf, int N, long long K,
int repeats = 7) {
    vector<double> vals;
    vals.reserve(repeats);
    for (int r = 0; r < repeats; ++r) {
        make_random_cycle(buf, N); // на каждый замер пересоздаём
случайный цикл, чтобы ломать локальность
        vals.push_back(measure_ticks_per_access_once(buf, N, K));
    }
    sort(vals.begin(), vals.end());
    return vals[repeats / 2];
}

int main() {
    srand((unsigned)time(nullptr));

#ifdef _WIN32
    // фиксируем процесс на одном ядре и поднимаем приоритет
    HANDLE hProc = GetCurrentProcess();
    HANDLE hThread = GetCurrentThread();

    DWORD_PTR mask = 1ull << 2;
    SetProcessAffinityMask(hProc, mask);
    SetThreadAffinityMask(hProc, mask);
    SetThreadAffinityMask(hThread, mask);

    SetPriorityClass(hProc, REALTIME_PRIORITY_CLASS);
    SetThreadPriority(hThread, THREAD_PRIORITY_TIME_CRITICAL);
#endif

    // разгон CPU ~1 сек
    {
        const int M = 512;
        double s = 0.0;
```

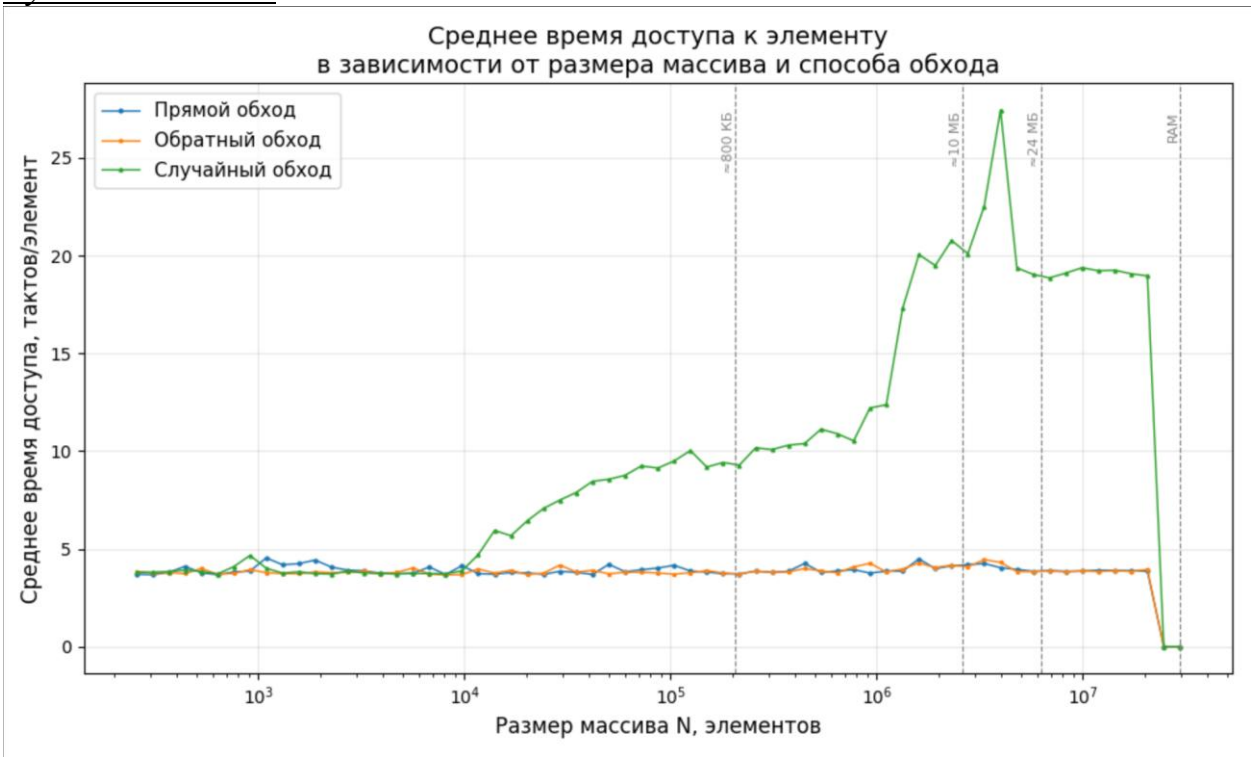
```
        clock_t t0 = clock();
        while (double(clock() - t0) / CLOCKS_PER_SEC < 1.0) {
            for (int i = 0; i < M; ++i)
                for (int j = 0; j < M; ++j)
                    s += i * j;
        }
        if ((int)s == 42) printf("turbo\n");
    }

    printf("#N\tforward\treverse\trandom (ticks/access)\n");

    int N = 256;

    // идём до ~256 МБ
    const int Nmax = 64 * 1024 * 1024;

    // диапазон обращений
    const long long min_accesses = 20'000'000LL;
    const long long max_accesses = 500'000'000LL;

    while (N <= Nmax) {
        int *a = (int*)malloc((size_t)N * sizeof(int));
        if (!a) {
            printf("malloc failed at N=%d\n", N);
            break;
        }

        // подбираем K так, чтобы N*K было в [min_accesses,
max_accesses]
        long long K = min_accesses / N;
        if (K < 1) K = 1;
        if (K * N > max_accesses)
            K = max_accesses / N;
        if (K < 1) K = 1;

        // прямой
        make_forward(a, N);
        double fwd = measure_ticks_per_access(a, N, K);

        // обратный
        make_backward(a, N);
        double bwd = measure_ticks_per_access(a, N, K);

        // случайный
        double rnd = measure_random_ticks_per_access(a, N, K);

        printf("%d\t%.3f\t%.3f\t%.3f\n", N, fwd, bwd, rnd);

        free(a);

        N = (int)floor(N * 1.2);
        if (N <= 0) N = 1;
    }

    return 0;
}
```

Приложение 2. *Код CMakeLists.txt*

```
cmake_minimum_required(VERSION 3.15)
project(lab5)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(lab5 src/main.cpp)

set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

if(MINGW)
    target_compile_options(lab5 PRIVATE -O1)
endif()

add_custom_target(show_flags ALL
        COMMAND ${CMAKE_CXX_COMPILER} -v
        COMMENT "Show compiler"
)

message(STATUS "CMAKE_CXX_FLAGS = ${CMAKE_CXX_FLAGS}")

if(MSVC)
    set_property(TARGET lab5 PROPERTY
            MSVC_RUNTIME_LIBRARY
"MultiThreaded$<$<CONFIG:Debug>:Debug>")

    target_link_libraries(lab5 PRIVATE
            ws2_32
            comctl32
            gdi32
            ole32
            oleaut32
            uuid
            vfw32
    )
endif()

if(MINGW)
    target_link_options(lab5 PRIVATE
            "-static"
            "-static-libgcc"
            "-static-libstdc++"
    )
endif()

if(APPLE)
    target_link_libraries(lab5 PRIVATE
            "-framework IOKit"
            "-framework Cocoa"
            "-framework OpenGL"
    )
endif()
```

Приложение 3. *Результаты измерений*

Ryzen 9 AI 365:



WSL Ubuntu on Ryzen 9 AI 365: