Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

Лабораторная работа №2 по курсу

«Операционные системы»

Группа: М8О-209БВ-24

Студент: Кобзев К. А.

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 10.10.25

Москва, 2025

Постановка задачи

Вариант 13.

Составить программу на языке Си, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы (Windows/Unix). Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска вашей программы.

Так же необходимо уметь продемонстрировать количество потоков, используемое вашей программой с помощью стандартных средств операционной системы.

В отчете привести исследование зависимости ускорения и эффективности алгоритма от входных данных и количества потоков. Получившиеся результаты необходимо объяснить.

13 вариант) Наложить K раз фильтр, использующий матрицу свертки, на матрицу, состоящую из вещественных чисел. Размер окна задается пользователем.

Общий метод и алгоритм решения

Использованные системные вызовы:

- pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg); — создает новый поток для выполнения задачи.
- pthread_join(pthread_t thread, void **retval); — блокирует вызывающий поток до тех пор, пока указанный поток не завершится. Используется как механизм синхронизации между итерациями.
- pthread_t — тип данных для хранения идентификатора потока.
- thread_data_t (пользовательская структура) – структура для передачи в поток всех необходимых ему данных: указателей на входную и выходную матрицы, размеров матриц и ядра, а также диапазона строк, который должен обработать данный поток.

Ключевые особенности алгоритма:

1. Параллелизация по данным: обработка матрицы разделяется между потоками. Каждый поток отвечает за вычисление значений для своего горизонтального участка (полосы) результирующей матрицы.
2. Итеративность: фильтр применяется K раз. Результат предыдущей итерации становится входными данными для следующей.
3. Двойная буферизация: для корректной работы итераций используются две матрицы: одна для чтения (входная), другая для записи (выходная). После каждой итерации указатели на них меняются местами. Это предотвращает "гонку данных", когда один поток читает значения, которые другой поток уже успел обновить в той же итерации.
4. Обработка границ: алгоритм свертки не применяется к крайним строкам и столбцам матрицы (шириной в ksize/2), так как для них ядро свертки выйдет за пределы исходных данных. Рабочая область для вычислений соответственно уменьшается.

Этапы работы программы:

1. Инициализация:
   - Принимает параметры из командной строки: размеры матрицы (rows, cols), размер ядра (ksize), количество итераций (K) и максимальное число потоков (num_threads).
   - Выделяет память под две матрицы (входную и результирующую) и ядро свертки.
   - Заполняет исходную матрицу и ядро случайными вещественными числами.
2. Применение свертки (цикл K раз):

- Рабочая область матрицы (исключая границы) делится на num_threads горизонтальных полос.
- Создается num_threads потоков. Каждому потоку передается структура thread_data_t с его уникальным диапазоном строк для обработки.
- Основной поток ожидает завершения всех рабочих потоков с помощью pthread_join. Этот вызов действует как барьер синхронизации, гарантируя, что вся матрица обработана перед началом следующей итерации.
- Указатели на входную и выходную матрицы меняются местами для подготовки к следующей итерации.

3. Завершение:
   - После выполнения всех итераций измеряется общее время работы.
   - Вся выделенная динамически память освобождается.

Эффективность алгоритма

Для анализа эффективности многопоточного, алгоритма были проведены замеры времени выполнения при количестве потоков от 1 до 16. Полученные данные о времени исполнения, а также рассчитанные на их основе ускорение и эффективность, представлены в таблице и на графиках ниже.

Running with 1 threads...

Time: 95.957 ms

Running with 2 threads...

Time: 49.092 ms

Running with 3 threads...

Time: 35.121 ms

Running with 4 threads...

Time: 26.989 ms

Running with 5 threads...

Time: 31.354 ms

Running with 6 threads...

Time: 27.679 ms

Running with 7 threads...

Time: 24.306 ms

Running with 8 threads...

Time: 23.041 ms

Running with 9 threads...

Time: 22.211 ms

Running with 10 threads...

Time: 21.092 ms

Running with 11 threads...

Time: 20.188 ms

Running with 12 threads...

Time: 20.293 ms

Running with 13 threads...

Time: 20.169 ms

Running with 14 threads...

Time: 19.362 ms

Running with 15 threads...

Time: 20.431 ms

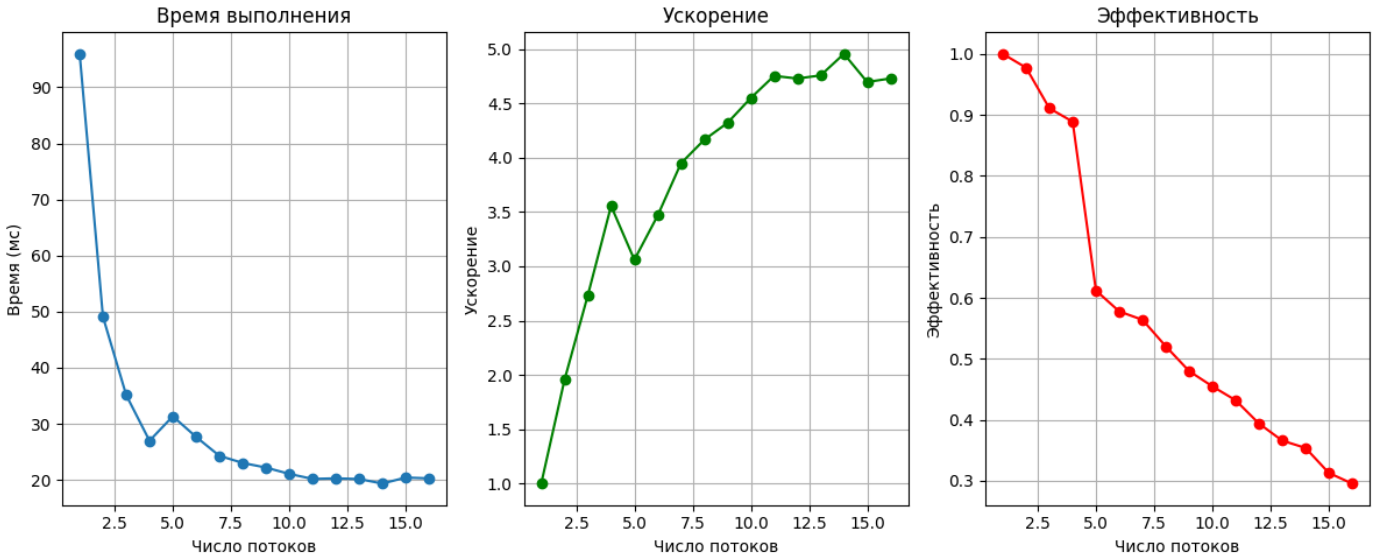
Running with 16 threads...

Time: 20.289 ms

Таблица результатов:

Потоки	Время (мс)	Ускорение	Эффективность
1	95.96	1.00	1.00
2	49.09	1.95	0.98
3	35.12	2.73	0.91
4	26.99	3.56	0.89
5	31.35	3.06	0.61
6	27.68	3.47	0.58
7	24.31	3.95	0.56
8	23.04	4.16	0.52
9	22.21	4.32	0.48
10	21.09	4.55	0.45
11	20.19	4.75	0.43
12	20.29	4.73	0.39
13	20.17	4.76	0.37
14	19.36	4.96	0.35
15	20.43	4.70	0.31

16 | 20.29 | 4.73 | 0.30



Анализ графиков и таблицы позволяет выделить несколько характерных этапов в поведении производительности системы.

1. Начальное ускорение (1-4 потока):

На этом участке наблюдается наиболее значительное падение времени выполнения. Ускорение растет почти линейно, а эффективность держится на очень высоком уровне (0.89 для 4 потоков). Это говорит о том, что задача эффективно распараллеливается, и накладные расходы на создание и управление потоками минимальны по сравнению с выгодой от параллельных вычислений на свободных ядрах процессора.

2. Аномалия и замедление роста (5-8 потоков):

При 5 потоках наблюдается аномальный скачок времени выполнения, что приводит к резкому падению ускорения и эффективности. Это может быть связано с особенностями планировщика операционной системы или архитектуры процессора, когда нечетное количество потоков распределяется по физическим ядрам неоптимально. Однако уже при 6-8 потоках производительность снова улучшается, хотя темп роста ускорения замедляется. Эффективность продолжает плавно снижаться, так как накладные расходы на синхронизацию и управление потоками становятся все более заметными.

3. Выход на плато производительности (9-16 потоков):

При дальнейшем увеличении числа потоков время выполнения стабилизируется в районе 20 мс. Прирост производительности становится минимальным или вовсе отсутствует. График ускорения выходит на плато, достигая своего пика (4.96) при 14 потоках. Дальнейшее увеличение числа потоков не дает выгоды и может даже незначительно ухудшать производительность (как видно при 15 и 16 потоках).

Это состояние насыщения объясняется тем, что количество потоков превысило число доступных для параллельной обработки вычислительных ресурсов (физических и/или логических ядер). В этот момент основными сдерживающими факторами становятся:

- Контекстное переключение: Операционная система тратит значительное время на переключение между потоками, отнимая ресурсы у полезных вычислений.

- Конкуренция за ресурсы: Множество потоков начинают конкурировать за общие ресурсы, такие как кэш-память процессора и пропускная способность шины памяти, что приводит к увеличению задержек.

Таким образом, эксперимент показывает, что для данной задачи и тестовой системы оптимальное количество потоков находится в диапазоне 11-14, а использование большего числа потоков нецелесообразно.

Код программы

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>
#include <string.h>

#define IDX(x, y, ncols) ((x) * (ncols) + (y))

typedef struct
{
    float *input, *output, *kernel;
    int rows, cols, ksize, start_row, end_row;
} thread_data_t;

void *convolve_part(void *arg)
{
    thread_data_t *data = (thread_data_t *)arg;
    int offset = data->ksize / 2;

    for (int i = data->start_row; i < data->end_row; i++)
    {
        for (int j = offset; j < data->cols - offset; j++)
```

```
        {
            float sum = 0.0f;
            for (int ki = -offset; ki <= offset; ki++)
            {
                for (int kj = -offset; kj <= offset; kj++)
                {
                    sum += data->input[IDX(i + ki, j + kj, data->cols)] *
                           data->kernel[IDX(ki + offset, kj + offset, data->ksize)];
                }
            }
            data->output[IDX(i, j, data->cols)] = sum;
        }
    }
    return NULL;
}

void apply_convolution(float *input, float *output, int rows, int cols, float *kernel, int ksize, int iterations, int num_threads)
{
    pthread_t threads[num_threads];
    thread_data_t thread_data[num_threads];

    int offset = ksize / 2;
    int work_rows = rows - 2 * offset;
    int rows_per_thread = work_rows / num_threads;

    float *current_input = input;
    float *current_output = output;

    for (int iter = 0; iter < iterations; iter++)
```

```
    {
        for (int t = 0; t < num_threads; t++)
        {
            int start = offset + t * rows_per_thread;
            int end = (t == num_threads - 1) ? (rows - offset) : (start + rows_per_thread);
            thread_data[t] = (thread_data_t){current_input, current_output, kernel, rows, cols, ksize,
start, end};
            pthread_create(&threads[t], NULL, convolve_part, &thread_data[t]);
        }
        for (int t = 0; t < num_threads; t++)
        {
            pthread_join(threads[t], NULL);
        }
        float *tmp = current_input;
        current_input = current_output;
        current_output = tmp;
    }

    if (iterations % 2 == 0)
    {
        memcpy(output, current_input, rows * cols * sizeof(float));
    }
}

double get_time_ms()
{
    struct timeval tv;
    gettimeofday(&tv, NULL);
    return (double)tv.tv_sec * 1000.0 + (double)tv.tv_usec / 1000.0;
}
```

```
int main(int argc, char *argv[])
{
    if (argc != 6)
    {
        printf("Usage: %s rows cols kernel_size iterations num_threads\n", argv[0]);
        return 1;
    }

    int rows = atoi(argv[1]);
    int cols = atoi(argv[2]);
    int ksize = atoi(argv[3]);
    int iterations = atoi(argv[4]);
    int num_threads = atoi(argv[5]);

    float *matrix = malloc(rows * cols * sizeof(float));
    float *result = malloc(rows * cols * sizeof(float));
    float *kernel = malloc(ksize * ksize * sizeof(float));

    if (!matrix || !result || !kernel)
    {
        perror("Failed to allocate memory");
        return 1;
    }

    for (int i = 0; i < rows * cols; i++)
        matrix[i] = (float)(rand() % 100) / 10.0f;
    for (int i = 0; i < ksize * ksize; i++)
        kernel[i] = 1.0f / (ksize * ksize);
```

```
    double start = get_time_ms();
    apply_convolution(matrix, result, rows, cols, kernel, ksize, iterations, num_threads);
    double end = get_time_ms();

    printf("Time taken: %.3f ms\n", end - start);

    free(matrix);
    free(result);
    free(kernel);
    return 0;
}
```

lab2.sh

```
#!/bin/bash

gcc -pthread main.c -o main

ROWS=2000
COLS=2000
KSIZE=5
ITERATIONS=10

echo "Starting tests with ROWS=$ROWS, COLS=$COLS, KSIZE=$KSIZE, ITERATIONS=$ITERATIONS"
echo "-------------------------------------------------"

for threads in 1 2 3 4 5 6 7 8 9 10
do
  echo "Thread №: $threads"
  echo "-------------------------------------------------"
```

```
  ./main $ROWS $COLS $KSIZE $ITERATIONS $threads &
  CURRENT_PROG_PID=$!
  echo "PID: $CURRENT_PROG_PID"

  while kill -0 $CURRENT_PROG_PID 2>/dev/null; do
    if [ -d "/proc/$CURRENT_PROG_PID/task" ]; then
      LIVE_THREADS=$(ls /proc/$CURRENT_PROG_PID/task | wc -l)
      echo "$(date +%T%3N) - PID $CURRENT_PROG_PID - Live threads: $LIVE_THREADS"
    else
      echo "$(date +%T%3N) - PID $CURRENT_PROG_PID"
      break
    fi
    sleep 0.1
  done

  wait $CURRENT_PROG_PID
  echo "Process $CURRENT_PROG_PID completed."

  echo "------------------------------------------------"
done

echo "The end."
```

Протокол работы программы

Тестирование:

.venv) ➜ src git:(main) ✗ bash lab2.sh

Starting tests with ROWS=2000, COLS=2000, KSIZE=5, ITERATIONS=10

Thread №: 1

PID: 65270

18:04:403N - PID 65270

Time taken: 1921.736 ms

Process 65270 completed.

Thread №: 2

PID: 65278

18:04:423N - PID 65278

Time taken: 1011.085 ms

Process 65278 completed.

Thread №: 3

PID: 65284

18:04:443N - PID 65284

Time taken: 719.797 ms

Process 65284 completed.

Thread №: 4

PID: 65286

18:04:443N - PID 65286

Time taken: 532.942 ms

Process 65286 completed.

Thread №: 5

PID: 65292

18:04:453N - PID 65292

Time taken: 519.305 ms

Process 65292 completed.

Thread №: 6

PID: 65294

18:04:453N - PID 65294

Time taken: 473.498 ms

Process 65294 completed.

Thread №: 7

PID: 65296

18:04:463N - PID 65296

Time taken: 462.534 ms

Process 65296 completed.

Thread №: 8

PID: 65300

18:04:463N - PID 65300

Time taken: 438.499 ms

Process 65300 completed.

Thread №: 9

PID: 65306

18:04:473N - PID 65306

Time taken: 417.290 ms

Process 65306 completed.

Thread №: 10

PID: 65308

18:04:473N - PID 65308

Time taken: 410.539 ms

Process 65308 completed.

The end.

strace

204 execve("./main", ["./main", "200", "200", "5", "1", "2"], 0xfffff3ec3410 /* 12 vars */) = 0

204 brk(NULL) = 0x6b35000

204 mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffa8a25000

204 faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)

204 openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3

204 newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=25959, ...}, AT_EMPTY_PATH) = 0

204 mmap(NULL, 25959, PROT_READ, MAP_PRIVATE, 3, 0) = 0xffffa8a1e000

204 close(3) = 0

204 openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3

204 read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0000y\2\0\0\0\0"..., 832) = 832

204 newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=1651408, ...}, AT_EMPTY_PATH) = 0

204 mmap(NULL, 1826912, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffa882d000

204 mmap(0xffffa8830000, 1761376, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0) = 0xffffa8830000

204 munmap(0xffffa882d000, 12288) = 0

204 munmap(0xffffa89df000, 49248) = 0

204 mprotect(0xffffa89b7000, 86016, PROT_NONE) = 0

204 mmap(0xffffa89cc000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x18c000) = 0xffffa89cc000

204 mmap(0xffffa89d2000, 49248, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xffffa89d2000

204 close(3) = 0

204 set_tid_address(0xffffa8a26050) = 204

204 set_robust_list(0xffffa8a26060, 24) = 0

204 rseq(0xffffa8a266a0, 0x20, 0, 0xd428bc00) = 0

204 mprotect(0xffffa89cc000, 16384, PROT_READ) = 0

204 mprotect(0x41f000, 4096, PROT_READ) = 0

204 mprotect(0xffffa8a2a000, 8192, PROT_READ) = 0

204 prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY}) = 0

204 munmap(0xffffa8a1e000, 25959) = 0

204 getrandom("\x78\x11\xcf\x13\x63\x2f\x5d\xaa", 8, GRND_NONBLOCK) = 8

204 brk(NULL) = 0x6b35000

204 brk(0x6b56000) = 0x6b56000

204 mmap(NULL, 163840, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffa8808000

204 mmap(NULL, 163840, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffa87e0000

204 rt_sigaction(SIGRT_1, {sa_handler=0xffffa88ac0a0, sa_mask=[], sa_flags=SA_ONSTACK|SA_RESTART|SA_SIGINFO}, NULL, 8) = 0

204 rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0

204 mmap(NULL, 8454144, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0xffffa7e00000

204 mprotect(0xffffa7e10000, 8388608, PROT_READ|PROT_WRITE) = 0

204 rt_sigprocmask(SIG_BLOCK, ~[], [], 8) = 0

204 clone(child_stack=0xffffa860ea60, flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID, parent_tid=[205], tls=0xffffa860f8e0, child_tidptr=0xffffa860f270) = 205

204 rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0

205 rseq(0xffffa860f8c0, 0x20, 0, 0xd428bc00 <unfinished ...>

204 mmap(NULL, 8454144, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0xffffa7400000

205 <... rseq resumed>) = 0

```
204   mprotect(0xffffa7410000, 8388608, PROT_READ|PROT_WRITE <unfinished ...>
205   set_robust_list(0xffffa860f280, 24 <unfinished ...>
204   <... mprotect resumed>)          = 0
205   <... set_robust_list resumed>)   = 0
204   rt_sigprocmask(SIG_BLOCK, ~[],  <unfinished ...>
205   rt_sigprocmask(SIG_SETMASK, [],  <unfinished ...>
204   <... rt_sigprocmask resumed>[], 8) = 0
205   <... rt_sigprocmask resumed>NULL, 8) = 0
204   clone(child_stack=0xffffa7c0ea60,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLONE_SY
SVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
parent_tid=[206], tls=0xffffa7c0f8e0, child_tidptr=0xffffa7c0f270) = 206
204   rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
204   futex(0xffffa860f270, FUTEX_WAIT_BITSET|FUTEX_CLOCK_REALTIME, 205, NULL,
FUTEX_BITSET_MATCH_ANY <unfinished ...>
206   rseq(0xffffa7c0f8c0, 0x20, 0, 0xd428bc00) = 0
206   set_robust_list(0xffffa7c0f280, 24) = 0
206   rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
205   rt_sigprocmask(SIG_BLOCK, ~[RT_1], NULL, 8) = 0
205   madvise(0xffffa7e00000, 8314880, MADV_DONTNEED) = 0
205   exit(0)                          = ?
204   <... futex resumed>)             = 0
205   +++ exited with 0 +++
204   futex(0xffffa7c0f270, FUTEX_WAIT_BITSET|FUTEX_CLOCK_REALTIME, 206, NULL,
FUTEX_BITSET_MATCH_ANY <unfinished ...>
206   rt_sigprocmask(SIG_BLOCK, ~[RT_1], NULL, 8) = 0
206   madvise(0xffffa7400000, 8314880, MADV_DONTNEED) = 0
206   exit(0)                          = ?
204   <... futex resumed>)             = 0
204   newfstatat(1, "",  <unfinished ...>
206   +++ exited with 0 +++
204   <... newfstatat resumed>{st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0), ...},
AT_EMPTY_PATH) = 0
204   write(1, "Time taken: 6.772 ms\n", 21) = 21
204   munmap(0xffffa8808000, 163840)    = 0
```

```
204   munmap(0xffffa87e0000, 163840)    = 0
204   exit_group(0)                     = ?
204   +++ exited with 0 +++
```

Вывод

В ходе выполнения лабораторной работы была успешно разработана и протестирована программа для многопоточного наложения фильтра свертки на матрицу вещественных чисел. Было продемонстрировано, что применение многопоточности позволяет значительно ускорить выполнение вычислительно интенсивной задачи свертки, особенно на больших объемах данных. Анализ результатов показал, что при чрезмерном увеличении числа потоков доминирующими факторами становятся накладные расходы на переключение контекста, синхронизацию и неэффективное использование кэш-памяти, что нивелирует все преимущества параллельных вычислений.