|  |  |  |  |
| --- | --- | --- | --- |
|  | | | |
| Федеральное государственное бюджетное  образовательное учреждение высшего образования «Новосибирский государственный технический университет» | | | |
|  | | | |
| Кафедра прикладной математики | | | |
|  | | | |
| Лабораторная работа № 3 | | | |
| по дисциплине «Параллельное программирование» | | | |
| **ПРОГРАММИРОВАНИЕ ВЗАИМОДЕЙСТВУЮЩИХ ПОТОКОВ** | | | |
|  | | | |
|  | Бригада 6 | ПМИ-71 Антонов С. |
|  | ПМИ-71 Арнольд э. |
|  | ПМИ-71 Кайль Д. |
|  |
|  |
| Преподаватель | Щукин Г. А.  Городничев М. А. |
|  | | | |
| Новосибирск | | | |

**Задание**

1. Модифицируйте программу, из *примера 19*, заменив **a[i] = b[i]** в теле цикла на **a[i] = f(b[i])**, где **f** – некоторая чистая функция, задающая достаточно сложное вычисление, чтобы время ее работы составляло порядка 1 секунды. Запустите в терминальном окне программу **top**, перейдите в режим отображения загрузки отдельных ядер вычислительной системы, нажав клавишу «**1**». В другом окне скомпилируйте модифицированную программу с игнорированием директив OpenMP и запустите ее. Проанализируйте загрузку ядер, отображаемую программой **top**.
2. Скомпилируйте программу с поддержкой OpenMP. Запустите программу. Проанализируйте отличие в загрузке ядер в программе **top**.
3. Проведите эксперименты с исключением опции **reduction** из *примера 2*, вынесением переменной id из класса *частных* в класс *разделяемых* переменных. Проанализируйте возникающие ошибочные ситуации.
4. Перепишите программу из *примера 2* без использования опции **reduction**, но с применением директивы **critical**.

**Задание для самостоятельной работы**

На основании примеров и пояснений в теоретической части разработайте программу для решения задачи: филтрация изображения.

**Программа**

*imageprocessor.c*

#include <stdio.h>

#include <string.h>

#if defined(\_OPENMP)

#include <omp.h>

#endif

#include "imageprocessor.h"

#define IMAGE\_BITNESS 32

#define BYTE\_SIZE  255

#define BYTE\_VALUE 8

#define PIXEL\_SIZE 4                // +1 байт для 32-х разрядного заголовка

// Функция обработки ошибок

#define throwErr(msg) do {          \

    fprintf(stderr, "%s\n", msg);   \

    exit(EXIT\_FAILURE);             \

} while (0)

#define max(x, y) ((x) > (y) ? (x) : (y))

#define min(x, y) ((x) < (y) ? (x) : (y))

// Корректировка цвета фильтра после преобразований

#define correctFilterColor(filter, color) \

    (min(max(((filter)->factor \* color + (filter)->bias), 0), BYTE\_SIZE))

// Данные потока для обработки

typedef struct Chunk {

    Color\*\* pixels;

    uint32\_t x0;

    uint32\_t x1;

    uint32\_t y0;

    uint32\_t y1;

} Chunk;

// Чтение заголовков файла

static void readBitmap(FILE\* fp, BMPImage\* image) {

    size\_t err = 0;

    // Избегаем прагм упаковки структур и невозможности переноса.

    // Читаем все данные, без чтения округлённого размера структур в памяти

    err = fread(&image->file\_header.type, sizeof image->file\_header.type, 1, fp);

    if (err == 0)

        throwErr("Error: reading file\_header.type!");

    err = fread(&image->file\_header.size, sizeof image->file\_header.size, 1, fp);

    if (err == 0)

        throwErr("Error: reading file\_header.size!");

    err = fread(&image->file\_header.reserved1, sizeof image->file\_header.reserved1, 1, fp);

    if (err == 0)

        throwErr("Error: reading file\_header.reserved1!");

    err = fread(&image->file\_header.reserved2, sizeof image->file\_header.reserved2, 1, fp);

    if (err == 0)

        throwErr("Error: reading file\_header.reserved2!");

    err = fread(&image->file\_header.offset\_bits, sizeof image->file\_header.offset\_bits, 1, fp);

    if (err == 0)

        throwErr("Error: reading file\_header.offset\_bits!");

    err = fread(&image->info\_header.size, sizeof image->info\_header.size, 1, fp);

    if (err == 0)

        throwErr("Error: reading info\_header.size!");

    err = fread(&image->info\_header.width, sizeof image->info\_header.width, 1, fp);

    if (err == 0)

        throwErr("Error: reading info\_header.width!");

    err = fread(&image->info\_header.height, sizeof image->info\_header.height, 1, fp);

    if (err == 0)

        throwErr("Error: reading info\_header.height!");

    err = fread(&image->info\_header.planes, sizeof image->info\_header.planes, 1, fp);

    if (err == 0)

        throwErr("Error: reading info\_header.planes!");

    err = fread(&image->info\_header.bits\_count, sizeof image->info\_header.bits\_count, 1, fp);

    if (err == 0)

        throwErr("Error: reading info\_header.bits\_count!");

    err = fread(&image->info\_header.compression, sizeof image->info\_header.compression, 1, fp);

    if (err == 0)

        throwErr("Error: reading info\_header.compression!");

    err = fread(&image->info\_header.size\_image, sizeof image->info\_header.size\_image, 1, fp);

    if (err == 0)

        throwErr("Error: reading info\_header.size\_image!");

    err = fread(&image->info\_header.x\_per\_meter, sizeof image->info\_header.x\_per\_meter, 1, fp);

    if (err == 0)

        throwErr("Error: reading info\_header.x\_per\_meter!");

    err = fread(&image->info\_header.y\_per\_meter, sizeof image->info\_header.y\_per\_meter, 1, fp);

    if (err == 0)

        throwErr("Error: reading info\_header.y\_per\_meter!");

    err = fread(&image->info\_header.color\_used, sizeof image->info\_header.color\_used, 1, fp);

    if (err == 0)

        throwErr("Error: reading info\_header.color\_used!");

    err = fread(&image->info\_header.color\_important, sizeof image->info\_header.color\_important, 1, fp);

    if (err == 0)

        throwErr("Error: reading info\_header.color\_important!");

}

// Чтение пиксельных данных

static void readData(FILE\* fp, BMPImage\* image) {

    image->data = (uint8\_t\*)malloc(sizeof(uint8\_t) \* image->info\_header.size\_image);

    if (image->data == NULL)

        throwErr("Error: data out of memmory!");

    fseek(fp, sizeof(uint8\_t) \* image->file\_header.offset\_bits, SEEK\_SET);

    size\_t err = fread(image->data, sizeof(uint8\_t), image->info\_header.size\_image, fp);

    if (err == 0)

        throwErr("Error: reading image->data!");

}

// Чтение изображения

void readImage(const char\* filename, BMPImage\* image) {

    if (image == NULL)

        throwErr("Error: image null ptr!");

    FILE\* fp = fopen(filename, "rb");

    if (fp == NULL)

        throwErr("Error: read file opening!");

    readBitmap(fp, image);

    readData(fp, image);

    fclose(fp);

}

// Запись заголовков файла

static void writeBitmap(FILE\* fp, const BMPImage\* image) {

    // Избегаем прагм упаковки структур и невозможности переноса.

    // Записываем все данные, без записи округлённого размера структур в памяти

    fwrite(&image->file\_header.type, sizeof(uint8\_t), sizeof image->file\_header.type, fp);

    fwrite(&image->file\_header.size, sizeof(uint8\_t), sizeof image->file\_header.size, fp);

    fwrite(&image->file\_header.reserved1, sizeof(uint8\_t), sizeof image->file\_header.reserved1, fp);

    fwrite(&image->file\_header.reserved2, sizeof(uint8\_t), sizeof image->file\_header.reserved2, fp);

    fwrite(&image->file\_header.offset\_bits, sizeof(uint8\_t), sizeof image->file\_header.offset\_bits, fp);

    fwrite(&image->info\_header.size, sizeof(uint8\_t), sizeof image->info\_header.size, fp);

    fwrite(&image->info\_header.width, sizeof(uint8\_t), sizeof image->info\_header.width, fp);

    fwrite(&image->info\_header.height, sizeof(uint8\_t), sizeof image->info\_header.height, fp);

    fwrite(&image->info\_header.planes, sizeof(uint8\_t), sizeof image->info\_header.planes, fp);

    fwrite(&image->info\_header.bits\_count, sizeof(uint8\_t), sizeof image->info\_header.bits\_count, fp);

    fwrite(&image->info\_header.compression, sizeof(uint8\_t), sizeof image->info\_header.compression, fp);

    fwrite(&image->info\_header.size\_image, sizeof(uint8\_t), sizeof image->info\_header.size\_image, fp);

    fwrite(&image->info\_header.x\_per\_meter, sizeof(uint8\_t), sizeof image->info\_header.x\_per\_meter, fp);

    fwrite(&image->info\_header.y\_per\_meter, sizeof(uint8\_t), sizeof image->info\_header.y\_per\_meter, fp);

    fwrite(&image->info\_header.color\_used, sizeof(uint8\_t), sizeof image->info\_header.color\_used, fp);

    fwrite(&image->info\_header.color\_important, sizeof(uint8\_t), sizeof image->info\_header.color\_important, fp);

}

// Запись пиксельных данных

static void writeData(FILE\* fp, const BMPImage\* image) {

    fseek(fp, sizeof(uint8\_t) \* image->file\_header.offset\_bits, SEEK\_SET);

    fwrite(image->data, sizeof(uint8\_t), image->info\_header.size\_image, fp);

}

// Запись изображения

void writeImage(const char\* filename, const BMPImage\* image) {

    if (image == NULL)

        throwErr("Error: image null ptr!");

    FILE\* fp = fopen(filename, "wb");

    if (fp == NULL)

        throwErr("Error: write file opening!");

    writeBitmap(fp, image);

    writeData(fp, image);

    fclose(fp);

}

// Копирование изображения

void copyImage(BMPImage\* image\_dest, const BMPImage\* image\_src) {

    if (image\_dest == NULL)

        throwErr("Error: dest null ptr!");

    if (image\_src == NULL)

        throwErr("Error: src null ptr!");

    image\_dest->file\_header = image\_src->file\_header;

    image\_dest->info\_header = image\_src->info\_header;

    image\_dest->data = (uint8\_t\*)malloc(sizeof(uint8\_t) \* image\_dest->info\_header.size\_image);

    if (image\_dest->data == NULL)

        throwErr("Error: dest data out of memmory!");

    memcpy(image\_dest->data, image\_src->data, sizeof(uint8\_t) \* image\_dest->info\_header.size\_image);

}

// Получить указатель пикселя из позиции в пиксельных данных

static uint8\_t\* getPixelPtr(BMPImage\* image, uint32\_t x, uint32\_t y) {

    uint8\_t\* pixel\_ptr = NULL;

    // Доступ к пикселю в пиксельных данных

    if (x < image->info\_header.width && y < image->info\_header.height) {

        uint32\_t row = 0 < image->info\_header.height ? image->info\_header.height - y - 1 : y;

        uint32\_t row\_size = ((image->info\_header.bits\_count \*

                              image->info\_header.width + IMAGE\_BITNESS - 1) / IMAGE\_BITNESS) \* PIXEL\_SIZE;

        uint32\_t bits\_per\_pixel = image->info\_header.bits\_count / BYTE\_VALUE;

        pixel\_ptr = row \* row\_size + x \* bits\_per\_pixel + image->data;

    }

    return pixel\_ptr;

}

// Изменить цвет пикселя

void setPixelColor(BMPImage\* image, uint32\_t x, uint32\_t y, Color pixel) {

    uint8\_t\* pixel\_ptr = getPixelPtr(image, x, y);

    if (pixel\_ptr == NULL)

        throwErr("Error: pixel abroad when set!");

    pixel\_ptr[0] = pixel.b;

    pixel\_ptr[1] = pixel.g;

    pixel\_ptr[2] = pixel.r;

}

// Получить цвет пикселя

Color getPixelColor(BMPImage\* image, uint32\_t x, uint32\_t y) {

    uint8\_t\* pixel\_ptr = getPixelPtr(image, x, y);

    if (pixel\_ptr == NULL)

        throwErr("Error: pixel abroad when get!");

    return (Color){pixel\_ptr[2], pixel\_ptr[1], pixel\_ptr[0]};

}

static void filterImageChunk(BMPImage\* image, const Filter\* filter, Chunk\* chunk) {

    // Фильтрация всех пикселей изображения

    for (uint32\_t x = chunk->x0; x < chunk->x1; ++x)

        for (uint32\_t y = chunk->y0; y < chunk->y1; ++y) {

            FilterColor filter\_color = (FilterColor){0, 0, 0};

            // Применение матрицы фильтра к пикселю и его соседям

            for (uint8\_t filter\_x = 0; filter\_x < filter->r; ++filter\_x)

                for (uint8\_t filter\_y = 0; filter\_y < filter->r; ++filter\_y) {

                    uint32\_t image\_x = (x - filter->r / 2 + filter\_x + image->info\_header.width) % image->info\_header.width;

                    uint32\_t image\_y = (y - filter->r / 2 + filter\_y + image->info\_header.height) % image->info\_header.height;

                    Color image\_pixel = getPixelColor(image, image\_x, image\_y);

                    filter\_color.r += image\_pixel.r \* filterAccess(filter, filter\_x, filter\_y);

                    filter\_color.g += image\_pixel.g \* filterAccess(filter, filter\_x, filter\_y);

                    filter\_color.b += image\_pixel.b \* filterAccess(filter, filter\_x, filter\_y);

                }

            // Сохранение преобразованного пикселя с выравниванием по размеру байта

            chunk->pixels[x][y].r = correctFilterColor(filter, filter\_color.r);

            chunk->pixels[x][y].g = correctFilterColor(filter, filter\_color.g);

            chunk->pixels[x][y].b = correctFilterColor(filter, filter\_color.b);

        }

    // Замещение изображения на преобразованное

    for (uint32\_t x = chunk->x0; x < chunk->x1; ++x)

        for (uint32\_t y = chunk->y0; y < chunk->y1; ++y)

            setPixelColor(image, x, y, chunk->pixels[x][y]);

}

// Фильтрация изображения

void filterImage(BMPImage\* image, const Filter\* filter, uint8\_t threads\_count) {

    #if defined(\_OPENMP)

    omp\_set\_dynamic(0);

    omp\_set\_num\_threads(threads\_count);

    #endif

    // Инициализация пиксельного поля для преобразованного изображения

    Color\*\* new\_pixels = (Color\*\*)malloc(sizeof(Color\*) \* image->info\_header.width);

    if (new\_pixels == NULL)

        throwErr("Error: new\_pixels out of memmory!");

    #pragma omp parallel for

    for (uint32\_t i = 0; i < image->info\_header.width; ++i) {

        new\_pixels[i] = (Color\*)malloc(sizeof(Color) \* image->info\_header.height);

        if (new\_pixels[i] == NULL)

            throwErr("Error: new\_pixels[i] out of memmory!");

    }

    // Инициализация блоков данных

    Chunk\* chunks = (Chunk\*)malloc(sizeof(Chunk) \* threads\_count);

    if (new\_pixels == NULL)

        throwErr("Error: chunks out of memmory!");

    uint32\_t chunk\_size = image->info\_header.width / threads\_count;

    uint8\_t remainder = image->info\_header.width % threads\_count;

    // Определение размеров блоков данных

    uint8\_t shift = 0;

    for (uint8\_t i = 0; i < threads\_count; ++i) {

        uint8\_t step = i < remainder ? 1 : 0;

        chunks[i].pixels = new\_pixels;

        chunks[i].x0 = i \* chunk\_size + shift;

        chunks[i].x1 = i \* chunk\_size + chunk\_size + shift + step;

        chunks[i].y0 = 0;

        chunks[i].y1 = image->info\_header.height;

        shift += step;

    }

    #pragma omp parallel

    {

        uint8\_t thread\_num = 0;

        #if defined(\_OPENMP)

        thread\_num = omp\_get\_thread\_num();

        #endif

        filterImageChunk(image, filter, &chunks[thread\_num]);

    }

    #pragma omp parallel for

    for (uint32\_t x = 0; x < image->info\_header.width; ++x)

        free(new\_pixels[x]);

    free(new\_pixels);

    free(chunks);

}

}

*imageprocessor.h*

#ifndef IMAGEPROCESSOR\_H

#define IMAGEPROCESSOR\_H

#include "filter.h"

// Цвет пикселя BMP изображения

typedef struct Color {

    uint8\_t r;

    uint8\_t g;

    uint8\_t b;

} Color;

typedef struct BitmapFileHeader {

    uint16\_t type;                  // Тип формата

    uint32\_t size;                  // Размер файла в байтах

    uint16\_t reserved1;

    uint16\_t reserved2;

    uint32\_t offset\_bits;           // Сдвиг данных, относительно начала

} BitmapFileHeader;

// Основная структура BMP файла

typedef struct BitmapInfoHeader {

    uint32\_t size;                  // Размер структуры в байтах

    uint32\_t width;                 // Ширина в пикселях

    uint32\_t height;                // Высота в пикселях

    uint16\_t planes;                // Для Windows: значки, курсор

    uint16\_t bits\_count;            // Количество бит на пиксель

    uint32\_t compression;           // Способ хранения пикселей

    uint32\_t size\_image;            // Размер пиксельных данных в байтах

    int32\_t  x\_per\_meter;           // Кол-во пикселей на метр горизонтали

    int32\_t  y\_per\_meter;           // Кол-во пикселей на метр по вертикали

    uint32\_t color\_used;            // Размер таблицы цветов в ячейках

    uint32\_t color\_important;       // Кол-во ячеек от начала таблицы цветов

} BitmapInfoHeader;

typedef struct BMPImage {

    BitmapFileHeader file\_header;

    BitmapInfoHeader info\_header;

    uint8\_t\* data;                  // Пиксельные данные.

} BMPImage;                         // Каждый пиксель представляет из себя 3 подряд идущих цвета: b, g, r

// Чтение/запись изображения

void readImage(const char\* filename, BMPImage\* image);

void writeImage(const char\* filename, const BMPImage\* image);

// Копирование изображение

void copyImage(BMPImage\* image\_dest, const BMPImage\* image\_src);

// Фильтрация изображения

void filterImage(BMPImage\* image, const Filter\* filter, uint8\_t threads\_count);

// Доступ к отдельным пикселям изображения

void setPixelColor(BMPImage\* image, uint32\_t x, uint32\_t y, Color pixel);

Color getPixelColor(BMPImage\* image, uint32\_t x, uint32\_t y);

#endif

*imageprocessor\_main.c*

#include <stdio.h>

#include <string.h>

#include <omp.h>

#include "imageprocessor.h"

#define DEMO\_ARGS\_COUNT 4

#define TEST\_ARGS\_COUNT 4

#define TEST\_RESULT\_FILENAME "result.txt"

#define TEST\_SIZE 4

static const char\* TEST\_FOLDER = "images/test\_images/";

static const char\* TEST\_FILENAMES[TEST\_SIZE] = {"2k.bmp", "4k.bmp", "5k.bmp", "8k.bmp"};

// Пресеты фильтров

typedef enum FilterNum {

    EMBOS,                  // Рельеф

    SHARPEN,                // Очертания

    EDGES,                  // Резкость

    BLUR,                   // Размытие

    FILTERNUM\_SIZE

} FilterNum;

// Поличить пресет фильтра по номеру

static Filter getFilter(uint8\_t filter\_num) {

    Filter filter;

    Filter const\_filter;

     switch (filter\_num) {

        case EMBOS: {

            const\_filter = filterCreate(1.0, 128.0, {

                -1, -1,  0,

                -1,  0,  1,

                 0,  1,  1

            });

            break;

        }

        case SHARPEN: {

            const\_filter = filterCreate(1.0, 0.0, {

                -1, -1, -1,

                -1,  9, -1,

                -1, -1, -1

            });

            break;

        }

        case EDGES: {

            const\_filter = filterCreate(1.0, 0.0, {

                0,  0, -1,  0,  0,

                0,  0, -1,  0,  0,

                0,  0,  2,  0,  0,

                0,  0,  0,  0,  0,

                0,  0,  0,  0,  0,

            });

            break;

        }

        case BLUR: {

            const\_filter = filterCreate(1.0 / 9.0, 0.0, {

                1, 0, 0, 0, 0, 0, 0, 0, 0,

                0, 1, 0, 0, 0, 0, 0, 0, 0,

                0, 0, 1, 0, 0, 0, 0, 0, 0,

                0, 0, 0, 1, 0, 0, 0, 0, 0,

                0, 0, 0, 0, 1, 0, 0, 0, 0,

                0, 0, 0, 0, 0, 1, 0, 0, 0,

                0, 0, 0, 0, 0, 0, 1, 0, 0,

                0, 0, 0, 0, 0, 0, 0, 1, 0,

                0, 0, 0, 0, 0, 0, 0, 0, 1

            });

            break;

        }

        default: {

            const\_filter = filterCreate(1.0, 0.0, {1});

            break;

        }

    }

    // Создание динамической матрицы, для её передачи из функции

    filter.r = const\_filter.r;

    filter.factor = const\_filter.factor;

    filter.bias = const\_filter.bias;

    filter.matrix = (double\*)malloc(sizeof(double) \* filter.r \* filter.r);

    if (filter.matrix == NULL) {

        fprintf(stderr, "Error: filter.matrix out of memmory!\n");

        exit(EXIT\_FAILURE);

    }

    memcpy(filter.matrix, const\_filter.matrix, sizeof(double) \* filter.r \* filter.r);

    return filter;

}

static char\* makeTestFilename(const char\* folder, const char\* filename) {

    size\_t result\_size = strlen(filename) + strlen(folder) + 1;

    char\* result\_filename = (char\*)malloc(sizeof(char\*) \* result\_size);

    snprintf(result\_filename, sizeof(char) \* result\_size, "%s%s", folder, filename);

    return result\_filename;

}

// Формирование результирующего названия файла по названию и суффиксу

static char\* makeResultFilename(const char\* filename, const char\* suffix) {

    size\_t result\_size = strlen(filename) + strlen(suffix) + 1;

    char\* result\_filename = (char\*)malloc(sizeof(char\*) \* result\_size);

    strncpy(result\_filename, filename, strlen(filename) - 4);

    result\_filename[strlen(filename) - 4] = '\0';

    strcat(result\_filename, suffix);

    strcat(result\_filename, ".bmp");

    return result\_filename;

}

static char\* getResultFilename(const char\* filename, uint8\_t filter\_num) {

    char\* result\_filename = NULL;

    switch (filter\_num) {

        case EMBOS:   result\_filename = makeResultFilename(filename, "-embos");   break;

        case SHARPEN: result\_filename = makeResultFilename(filename, "-sharpen"); break;

        case EDGES:   result\_filename = makeResultFilename(filename, "-edges");   break;

        case BLUR:    result\_filename = makeResultFilename(filename, "-blur");    break;

        default:      result\_filename = makeResultFilename(filename, "-none");    break;

    }

    return result\_filename;

}

static void demonstration(int argc, char\* argv[]) {

    if (argc < DEMO\_ARGS\_COUNT) {

        fprintf(stderr, "Wrong number of arguments!\n");

        fprintf(stderr, "Enter: <filename> <filter number> <threads count>\n");

        fprintf(stderr, "(Filter number: 0 - embos, 1 - edges, 2 - sharpen, 3 - blur)\n");

        exit(EXIT\_FAILURE);

    }

    const char\* filename = argv[1];

    uint8\_t filter\_num = atoi(argv[2]);

    uint8\_t threads\_count = atoi(argv[3]);

    if (FILTERNUM\_SIZE <= filter\_num) {

        fprintf(stderr, "Error: invalid filter number!\n");

        exit(EXIT\_FAILURE);

    }

    if (threads\_count == 0) {

        fprintf(stderr, "Error: the number of threads cannot be equal 0!\n");

        exit(EXIT\_FAILURE);

    }

    BMPImage image;

    readImage(filename, &image);

    Filter filter = getFilter(filter\_num);

    double start = omp\_get\_wtime();

    filterImage(&image, &filter, threads\_count);

    double stop = omp\_get\_wtime();

    char\* result\_filename = getResultFilename(filename, filter\_num);

    writeImage(result\_filename, &image);

    printf("\nElapsed time: %lf\n", stop - start);

    free(result\_filename);

    free(filter.matrix);

    free(image.data);

}

static void test(int argc, char\* argv[]) {

    if (argc < TEST\_ARGS\_COUNT) {

        fprintf(stderr, "Wrong number of arguments!\n");

        fprintf(stderr, "Enter: <filter num> <threads count> <measure count>\n");

        fprintf(stderr, "(Filter number: 0 - embos, 1 - edges, 2 - sharpen, 3 - blur)\n");

        exit(EXIT\_FAILURE);

    }

    uint8\_t filter\_num = atoi(argv[1]);

    uint8\_t threads\_count = atoi(argv[2]);

    uint8\_t measure\_count = atoi(argv[3]);

    FILE\* fp = fopen(TEST\_RESULT\_FILENAME, "w");

    printf("Program execution...\n");

    fprintf(fp, "size:\tthreads: 1\tthreads: 2\tthreads: 3\tthreads: 4\n");

    Filter filter = getFilter(filter\_num);

    for (uint8\_t test\_num = 0; test\_num != TEST\_SIZE; ++test\_num) {

        char\* test\_name = makeTestFilename(TEST\_FOLDER, TEST\_FILENAMES[test\_num]);

        BMPImage image;

        readImage(test\_name, &image);

        fprintf(fp, "%u x %u\t", image.info\_header.width, image.info\_header.height);

        for (uint8\_t i = 0; i != threads\_count; ++i) {

            double elapsed\_time = 0;

            for (uint8\_t j = 0; j != measure\_count; ++j) {

                BMPImage test\_image;

                copyImage(&test\_image, &image);

                double start = omp\_get\_wtime();

                filterImage(&test\_image, &filter, i + 1);

                double stop = omp\_get\_wtime();

                elapsed\_time += stop - start;

                free(test\_image.data);

            }

            fprintf(fp, "%lf\t", elapsed\_time / measure\_count);

        }

        free(test\_name);

        free(image.data);

        printf("Test %hhu complete\n", test\_num + 1);

        fprintf(fp, "\n");

    }

    free(filter.matrix);

    printf("Done.\n");

    fclose(fp);

}

int main(int argc, char\* argv[]) {

    #ifdef TEST

    test(argc, argv);

    #else

    demonstration(argc, argv);

    #endif

    return 0;

}

*Makefile*

DEFINES = -D\_POSIX\_C\_SOURCE -D\_BSD\_SOURCE

LIBS    = -lm -fopenmp

TARGET  = imageprocessor

CF\_DEMO = -std=c99 -O2 -g $(DEFINES)

CF\_TEST = -std=c99 -O2 -g $(DEFINES) -DTEST

demo: CFLAGS = $(CF\_DEMO)

demo: $(TARGET) cleanTemp

test: CFLAGS = $(CF\_TEST)

test: $(TARGET) cleanTemp

$(TARGET): $(TARGET)\_main.o $(TARGET).o

    gcc $(CFLAGS) -o $(TARGET) $(TARGET)\_main.o $(TARGET).o $(LIBS)

$(TARGET)\_main.o: $(TARGET)\_main.c $(TARGET).h

    gcc $(CFLAGS) -c $(TARGET)\_main.c $(LIBS)

$(TARGET).o: $(TARGET).c $(TARGET).h

    gcc $(CFLAGS) -c $(TARGET).c $(LIBS)

cleanTemp:

    rm -rf \*.o

clean:

    rm -rf $(TARGET)

*filter.h*

#ifndef FILTER\_H

#define FILTER\_H

#include <stdlib.h>

#include <stdint.h>

#include <math.h>

// Создание прямоугольного сглаживающего фильтра (const double[])

// Входные аргументы: коэффициент сглаживания, смещение, матрица фильтра

#define filterCreate(factor\_init, bias\_init, ...) ({        \

    Filter filter;                                          \

    static const double const\_data[] = \_\_VA\_ARGS\_\_;         \

    filter.matrix = (double[]) \_\_VA\_ARGS\_\_;                 \

    filter.r = sqrt(sizeof(const\_data) / sizeof(double));   \

    filter.factor = factor\_init;                            \

    filter.bias = bias\_init;                                \

    filter;                                                 \

})

// Доступ к элементу квадратной матрицы фильтра

#define filterAccess(filter, i, j) \

    ((filter)->matrix[(i) \* (filter)->r + (j)])

// Цвет ячейки фильтра

typedef struct FilterColor {

    double r;

    double g;

    double b;

} FilterColor;

typedef struct Filter {

    double\* matrix;         // Квадратная матрица фильтра

    uint8\_t r;              // Радиус

    double factor;          // Коэффициент сглаживания

    double bias;            // Смещение

} Filter;

#endif

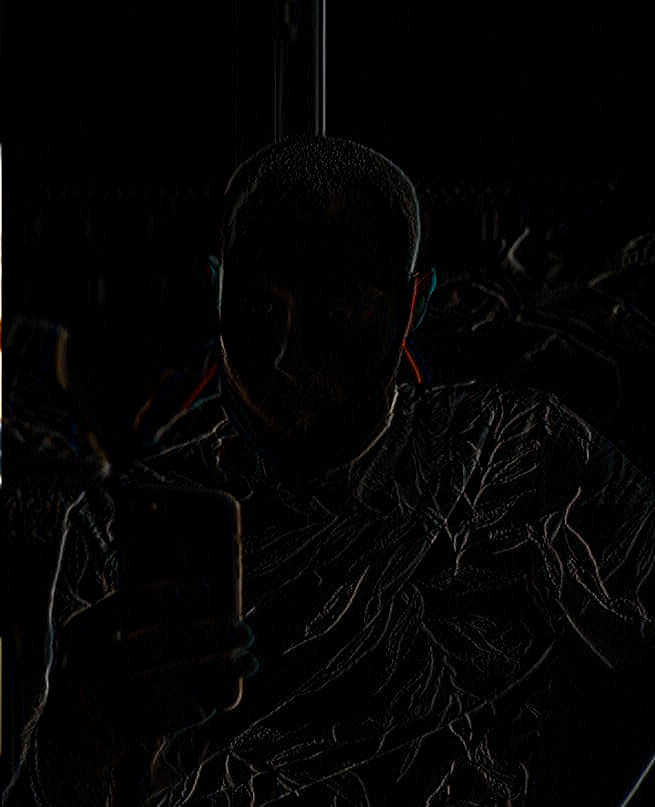
Результат работы программы:

Оригинальное изображение



Размытие

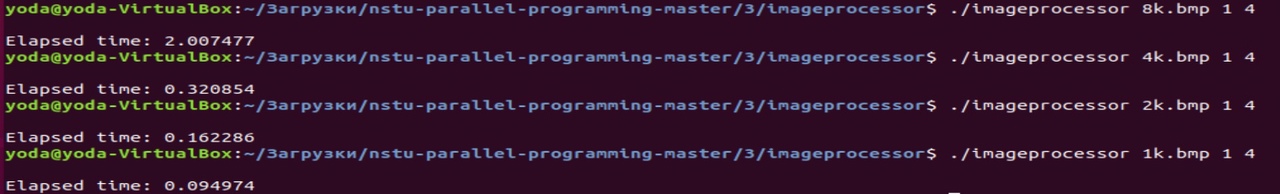


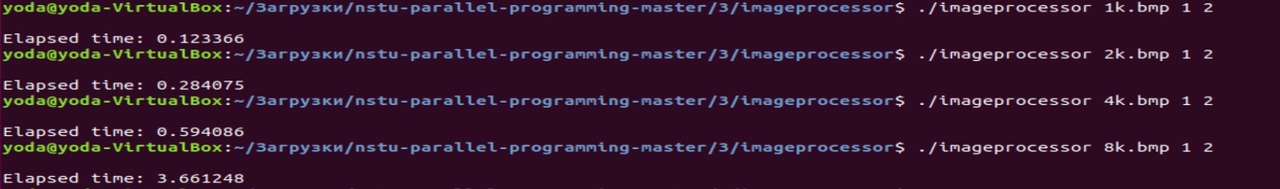
Очертание 

Рельеф



Резкость





Вывод

Мы познакомились с базовыми возможностями многопоточного программирования, научились работать с потоками , не имеющими информационных зависимостей