**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

Федеральное государственное бюджетное образовательное учреждение   
высшего образования

**«Сибирский государственный университет науки и технологий   
имени академика М.Ф. Решетнева»**

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

**ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ**

Параллельные вычисления

|  |
| --- |
| Алгоритмы свертки в OpenMP |

Руководитель А.Г. Зотин

подпись, дата инициалы, фамилия

Обучающийся БПИ20-02, 201219047 Р.А. Сухачев

номер группы, зачетной книжки подпись, дата инициалы, фамилия

Красноярск 2022 г.

# ЦЕЛЬ РАБОТЫ

Изучить различные алгоритмы выполнения линейной фильтрации на примере среднеарифметического фильтра и фильтра Гаусса с его различными аппроксимациями. Получить навыки разработки параллельных алгоритмов, реализующих линейные фильтры, принимая во внимания подходы к оптимизации обработки.

# порядок выполнения работы

1. Ознакомиться с теоретическими сведениями.

2. Ознакомится с алгоритмами линейной фильтрации изображения: линейный усредняющий фильтр, фильтр Гаусса.

3. Выполнить задания 1–2.

4. Подготовить отчет по лабораторной работе.

5.  Защитить лабораторную работу перед преподавателем.

# постановка задачи

Вариант 20

Задание 4.1. Разработайте консольное приложение, реализующее выполнение линейной фильтрации изображения. Необходимо разработать и реализовать последовательные и параллельные варианты среднеарифметического фильтра и фильтра Гаусса. Для среднеарифметического фильтра также необходимо реализовать быстрый вариант.

В приложение должен загружаться файл input.bmp и формироваться выходные файлы output\_alg\_var.bmp, где alg означает имя используемого алгоритма обработки (или его номер), а вар – варианта реализации.

Задание 4.2. Проведите экспериментальное исследование по обработке данных с разными реализациями алгоритмов фильтрации (не менее 20 запусков для каждой реализации). При применении алгоритмов получить показатели для разных размеров окрестности Ksize×Ksize для четырех изображений с разрешением от 1280×720 до 3840×2160. В рамках лабораторной работы Ksize = (RH×2+1) = (RW×2+1). Результаты исследования оформить в виде таблицы.

# ХОД РАБОТЫ

Задание 4.1/4.2.

Код программы:

#include <iostream>

#include <fstream>

#include <omp.h>

#include <math.h>

#include <cstring>

#include <string>

#include "BMPFileRW.h"

using namespace std;

#pragma warning(disable : 4996)

# define M\_PI 3.14159265358979323846

typedef double(\*TestFunctTemp1)(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize);

double medianLinearFilter(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

for (int y = 0; y < height; ++y)

{

for (int x = 0; x < width; ++x)

{

int LinF\_Value\_R = 0, LinF\_Value\_G = 0, LinF\_Value\_B = 0;

int Summ\_Value\_R = 0, Summ\_Value\_G = 0, Summ\_Value\_B = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0)

ky = 0;

if (ky > height - 1)

ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0)

kx = 0;

if (kx > width - 1)

kx = width - 1;

Summ\_Value\_R += rgb\_in[ky][kx].rgbtRed;

Summ\_Value\_G += rgb\_in[ky][kx].rgbtGreen;

Summ\_Value\_B += rgb\_in[ky][kx].rgbtBlue;

}

}

LinF\_Value\_R = Summ\_Value\_R / size;

LinF\_Value\_G = Summ\_Value\_G / size;

LinF\_Value\_B = Summ\_Value\_B / size;

rgb\_out[y][x].rgbtRed = LinF\_Value\_R;

rgb\_out[y][x].rgbtGreen = LinF\_Value\_G;

rgb\_out[y][x].rgbtBlue = LinF\_Value\_B;

}

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double medianLinearFilterParallel(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

#pragma omp parallel for

for (int y = 0; y < height; ++y)

{

for (int x = 0; x < width; ++x)

{

int LinF\_Value\_R = 0, LinF\_Value\_G = 0, LinF\_Value\_B = 0;

int Summ\_Value\_R = 0, Summ\_Value\_G = 0, Summ\_Value\_B = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0)

ky = 0;

if (ky > height - 1)

ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0)

kx = 0;

if (kx > width - 1)

kx = width - 1;

Summ\_Value\_R += rgb\_in[ky][kx].rgbtRed;

Summ\_Value\_G += rgb\_in[ky][kx].rgbtGreen;

Summ\_Value\_B += rgb\_in[ky][kx].rgbtBlue;

}

}

LinF\_Value\_R = Summ\_Value\_R / size;

LinF\_Value\_G = Summ\_Value\_G / size;

LinF\_Value\_B = Summ\_Value\_B / size;

rgb\_out[y][x].rgbtRed = LinF\_Value\_R;

rgb\_out[y][x].rgbtGreen = LinF\_Value\_G;

rgb\_out[y][x].rgbtBlue = LinF\_Value\_B;

}

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double\*\* gaussMatrix(int size) {

int RH = size / 2, RW = size / 2;

double\*\* matrix = new double\* [size];

for (int i = 0; i < size; i++)

matrix[i] = new double[size];

double SUM = 0;

for (int y = -RH; y <= RH; y++) {

for (int x = -RW; x <= RW; x++) {

int YK = y + RH;

int XK = x + RW;

double CF = (1 / (2 \* M\_PI \* pow(RH, 2))) \* exp(-1 \* (pow(x, 2) +

pow(y, 2)) / (2 \* pow(RH, 2)));

matrix[YK][XK] = CF;

SUM += CF;

}

}

for (int y = -RH; y <= RH; y++) {

for (int x = -RW; x <= RW; x++) {

int YK = y + RH;

int XK = x + RW;

matrix[YK][XK] /= SUM;

}

}

return matrix;

}

double medianGaussFilter(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

double\*\* matrix = gaussMatrix(ksize);

for (int y = 0; y < height; ++y)

{

for (int x = 0; x < width; ++x)

{

int LinF\_Value\_R = 0, LinF\_Value\_G = 0, LinF\_Value\_B = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0)

ky = 0;

if (ky > height - 1)

ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0)

kx = 0;

if (kx > width - 1)

kx = width - 1;

LinF\_Value\_R += rgb\_in[ky][kx].rgbtRed \*

matrix[dy + RH][dx + RW];

LinF\_Value\_G += rgb\_in[ky][kx].rgbtGreen \*

matrix[dy + RH][dx + RW];

LinF\_Value\_B += rgb\_in[ky][kx].rgbtBlue \*

matrix[dy + RH][dx + RW];

}

}

LinF\_Value\_R = LinF\_Value\_R < 0 ? 0 : LinF\_Value\_R;

LinF\_Value\_R = LinF\_Value\_R > 255 ? 255 : LinF\_Value\_R;

rgb\_out[y][x].rgbtRed = LinF\_Value\_R;

LinF\_Value\_G = LinF\_Value\_G < 0 ? 0 : LinF\_Value\_G;

LinF\_Value\_G = LinF\_Value\_G > 255 ? 255 : LinF\_Value\_G;

rgb\_out[y][x].rgbtGreen = LinF\_Value\_G;

LinF\_Value\_B = LinF\_Value\_B < 0 ? 0 : LinF\_Value\_B;

LinF\_Value\_B = LinF\_Value\_B > 255 ? 255 : LinF\_Value\_B;

rgb\_out[y][x].rgbtBlue = LinF\_Value\_B;

}

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double\*\* gaussMatrixParallel(int size) {

int RH = size / 2, RW = size / 2;

double\*\* matrix = new double\* [size];

for (int i = 0; i < size; i++)

matrix[i] = new double[size];

double SUM = 0;

#pragma omp parallel for reduction(+:SUM)

for (int y = -RH; y <= RH; y++) {

for (int x = -RW; x <= RW; x++) {

int YK = y + RH;

int XK = x + RW;

double CF = (1 / (2 \* M\_PI \* pow(RH, 2))) \* exp(-1 \* (pow(x, 2) +

pow(y, 2)) / (2 \* pow(RH, 2)));

matrix[YK][XK] = CF;

SUM += CF;

}

}

#pragma omp parallel for

for (int y = -RH; y <= RH; y++) {

for (int x = -RW; x <= RW; x++) {

int YK = y + RH;

int XK = x + RW;

matrix[YK][XK] /= SUM;

}

}

return matrix;

}

double medianGaussFilterParallel(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

double\*\* matrix = gaussMatrixParallel(ksize);

#pragma omp parallel for

for (int y = 0; y < height; ++y)

{

for (int x = 0; x < width; ++x)

{

int LinF\_Value\_R = 0, LinF\_Value\_G = 0, LinF\_Value\_B = 0;

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0)

ky = 0;

if (ky > height - 1)

ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0)

kx = 0;

if (kx > width - 1)

kx = width - 1;

LinF\_Value\_R += rgb\_in[ky][kx].rgbtRed \*

matrix[dy + RH][dx + RW];

LinF\_Value\_G += rgb\_in[ky][kx].rgbtGreen \*

matrix[dy + RH][dx + RW];

LinF\_Value\_B += rgb\_in[ky][kx].rgbtBlue \*

matrix[dy + RH][dx + RW];

}

}

LinF\_Value\_R = LinF\_Value\_R < 0 ? 0 : LinF\_Value\_R;

LinF\_Value\_R = LinF\_Value\_R > 255 ? 255 : LinF\_Value\_R;

rgb\_out[y][x].rgbtRed = LinF\_Value\_R;

LinF\_Value\_G = LinF\_Value\_G < 0 ? 0 : LinF\_Value\_G;

LinF\_Value\_G = LinF\_Value\_G > 255 ? 255 : LinF\_Value\_G;

rgb\_out[y][x].rgbtGreen = LinF\_Value\_G;

LinF\_Value\_B = LinF\_Value\_B < 0 ? 0 : LinF\_Value\_B;

LinF\_Value\_B = LinF\_Value\_B > 255 ? 255 : LinF\_Value\_B;

rgb\_out[y][x].rgbtBlue = LinF\_Value\_B;

}

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double medianQuickLinearFilter(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

for (int y = 0; y < height; ++y)

{

int Summ\_Value\_R = 0, Summ\_Value\_G = 0, Summ\_Value\_B = 0;

for (int x = 0; x < width; ++x)

{

int LinF\_Value\_R = 0, LinF\_Value\_G = 0, LinF\_Value\_B = 0;

if (x == 0) {

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0)

ky = 0;

if (ky > height - 1)

ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0)

kx = 0;

if (kx > width - 1)

kx = width - 1;

Summ\_Value\_R += rgb\_in[ky][kx].rgbtRed;

Summ\_Value\_G += rgb\_in[ky][kx].rgbtGreen;

Summ\_Value\_B += rgb\_in[ky][kx].rgbtBlue;

}

}

LinF\_Value\_R = Summ\_Value\_R / size;

LinF\_Value\_G = Summ\_Value\_G / size;

LinF\_Value\_B = Summ\_Value\_B / size;

rgb\_out[y][x].rgbtRed = LinF\_Value\_R;

rgb\_out[y][x].rgbtGreen = LinF\_Value\_G;

rgb\_out[y][x].rgbtBlue = LinF\_Value\_B;

}

else if (x > 0) {

int kx1 = x - RW - 1;

if (kx1 < 0) {

kx1 = 0;

}

int kx2 = x + RW;

if (kx2 > width - 1) {

kx2 = width - 1;

}

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0)

ky = 0;

if (ky > height - 1)

ky = height - 1;

Summ\_Value\_R -= rgb\_in[ky][kx1].rgbtRed;

Summ\_Value\_G -= rgb\_in[ky][kx1].rgbtGreen;

Summ\_Value\_B -= rgb\_in[ky][kx1].rgbtBlue;

Summ\_Value\_R += rgb\_in[ky][kx2].rgbtRed;

Summ\_Value\_G += rgb\_in[ky][kx2].rgbtGreen;

Summ\_Value\_B += rgb\_in[ky][kx2].rgbtBlue;

}

LinF\_Value\_R = Summ\_Value\_R / size;

LinF\_Value\_G = Summ\_Value\_G / size;

LinF\_Value\_B = Summ\_Value\_B / size;

rgb\_out[y][x].rgbtRed = LinF\_Value\_R;

rgb\_out[y][x].rgbtGreen = LinF\_Value\_G;

rgb\_out[y][x].rgbtBlue = LinF\_Value\_B;

}

}

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double medianQuickLinearFilterParallel(RGBTRIPLE\*\* rgb\_in, RGBTRIPLE\*\* rgb\_out, int height, int width, int ksize) {

double time\_start = omp\_get\_wtime();

int size = ksize \* ksize;

int RH = ksize / 2, RW = ksize / 2;

#pragma omp parallel for

for (int y = 0; y < height; ++y)

{

int Summ\_Value\_R = 0, Summ\_Value\_G = 0, Summ\_Value\_B = 0;

for (int x = 0; x < width; ++x)

{

int LinF\_Value\_R = 0, LinF\_Value\_G = 0, LinF\_Value\_B = 0;

if (x == 0) {

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0)

ky = 0;

if (ky > height - 1)

ky = height - 1;

for (int dx = -RW; dx <= RW; dx++) {

int kx = x + dx;

if (kx < 0)

kx = 0;

if (kx > width - 1)

kx = width - 1;

Summ\_Value\_R += rgb\_in[ky][kx].rgbtRed;

Summ\_Value\_G += rgb\_in[ky][kx].rgbtGreen;

Summ\_Value\_B += rgb\_in[ky][kx].rgbtBlue;

}

}

LinF\_Value\_R = Summ\_Value\_R / size;

LinF\_Value\_G = Summ\_Value\_G / size;

LinF\_Value\_B = Summ\_Value\_B / size;

rgb\_out[y][x].rgbtRed = LinF\_Value\_R;

rgb\_out[y][x].rgbtGreen = LinF\_Value\_G;

rgb\_out[y][x].rgbtBlue = LinF\_Value\_B;

}

else if (x > 0) {

int kx1 = x - RW - 1;

if (kx1 < 0) {

kx1 = 0;

}

int kx2 = x + RW;

if (kx2 > width - 1) {

kx2 = width - 1;

}

for (int dy = -RH; dy <= RH; dy++) {

int ky = y + dy;

if (ky < 0)

ky = 0;

if (ky > height - 1)

ky = height - 1;

Summ\_Value\_R -= rgb\_in[ky][kx1].rgbtRed;

Summ\_Value\_G -= rgb\_in[ky][kx1].rgbtGreen;

Summ\_Value\_B -= rgb\_in[ky][kx1].rgbtBlue;

Summ\_Value\_R += rgb\_in[ky][kx2].rgbtRed;

Summ\_Value\_G += rgb\_in[ky][kx2].rgbtGreen;

Summ\_Value\_B += rgb\_in[ky][kx2].rgbtBlue;

}

LinF\_Value\_R = Summ\_Value\_R / size;

LinF\_Value\_G = Summ\_Value\_G / size;

LinF\_Value\_B = Summ\_Value\_B / size;

rgb\_out[y][x].rgbtRed = LinF\_Value\_R;

rgb\_out[y][x].rgbtGreen = LinF\_Value\_G;

rgb\_out[y][x].rgbtBlue = LinF\_Value\_B;

}

}

}

double time\_stop = omp\_get\_wtime();

return time\_stop - time\_start;

}

double testMedianLinearFilter(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianLinearFilter(rgb\_in, rgb\_out, height, width, ksize);

}

double testMedianGaussFilter(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianGaussFilter(rgb\_in, rgb\_out, height, width, ksize);

}

double testMedianQuickLinearFilter(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianQuickLinearFilter(rgb\_in, rgb\_out, height, width, ksize);

}

double testMedianGaussFilterParallel(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianGaussFilterParallel(rgb\_in, rgb\_out, height, width, ksize);

}

double testMedianLinearFilterParallel(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianLinearFilterParallel(rgb\_in, rgb\_out, height, width, ksize);

}

double testMedianQuickLinearFilterParallel(RGBTRIPLE\*\*& rgb\_in, RGBTRIPLE\*\*& rgb\_out, int& height, int& width, int& ksize) {

return medianQuickLinearFilterParallel(rgb\_in, rgb\_out, height, width, ksize);

}

char\* inBMP(int i) {

string str;

str = "C:\\Users\\whoami\\source\\repos\\Parallel Computing\\lab\_4\\input\_X.bmp";

char\* cstr;

switch (i)

{

case 1:

str[60] = '1';

break;

case 2:

str[60] = '2';

break;

case 3:

str[60] = '3';

break;

case 4:

str[60] = '4';

break;

default:

break;

}

cstr = new char[str.length() + 1];

strcpy(cstr, str.c\_str());

return cstr;

}

char\* outBMP(int i, int alg) {

string str;

str = "C:\\Users\\whoami\\source\\repos\\Parallel Computing\\lab\_4\\output\_X\_Y.bmp";

char\* cstr;

switch (i)

{

case 1:

str[61] = '1';

break;

case 2:

str[61] = '2';

break;

case 3:

str[61] = '3';

break;

case 4:

str[61] = '4';

break;

default:

break;

}

switch (alg)

{

case 0:

str[63] = '1';

break;

case 1:

str[63] = '2';

break;

case 2:

str[63] = '3';

break;

case 3:

str[63] = '4';

break;

case 4:

str[63] = '5';

break;

case 5:

str[63] = '6';

break;

default:

break;

}

cstr = new char[str.length() + 1];

strcpy(cstr, str.c\_str());

cout << cstr << endl;

return cstr;

}

double AvgTrustedInterval(double& avg, double\*& times, int& cnt)

{

double sd = 0, newAVg = 0;

int newCnt = 0;

for (int i = 0; i < cnt; i++)

{

sd += (times[i] - avg) \* (times[i] - avg);

}

sd /= (cnt - 1.0);

sd = sqrt(sd);

for (int i = 0; i < cnt; i++)

{

if (avg - sd <= times[i] && times[i] <= avg + sd)

{

newAVg += times[i];

newCnt++;

}

}

if (newCnt == 0) newCnt = 1;

return newAVg / newCnt;

}

double TestIter(void\* Funct, RGBTRIPLE\*\* rgb\_in, int Height, int Width, int ksize, int iterations, int i, int alg)

{

double curtime = 0, avgTime = 0, avgTimeT = 0, correctAVG = 0;;

double\* Times = new double[iterations];

cout << endl;

RGBTRIPLE\*\* rgb\_out;

rgb\_out = new RGBTRIPLE \* [Height];

rgb\_out[0] = new RGBTRIPLE[Width \* Height];

for (int j = 1; j < Height; j++)

{

rgb\_out[j] = &rgb\_out[0][Width \* j];

}

for (int j = 0; j < iterations; j++)

{

curtime = ((\*(TestFunctTemp1)Funct)(rgb\_in, rgb\_out, Height, Width, ksize)) \* 1000;

Times[j] = curtime;

avgTime += curtime;

cout << "+";

}

cout << endl;

avgTime /= iterations;

cout << "AvgTime:" << avgTime << endl;

avgTimeT = AvgTrustedInterval(avgTime, Times, iterations);

cout << "AvgTimeTrusted:" << avgTimeT << endl;

char\* cstr = outBMP(i, alg);

BMPWrite(rgb\_out, Width, Height, cstr);

delete[] rgb\_out[0];

delete[] rgb\_out;

delete[] cstr;

return avgTimeT;

}

void test\_functions(void\*\* Functions, string(&function\_names)[6])

{

RGBTRIPLE\*\* rgb\_in;

BITMAPFILEHEADER header;

BITMAPINFOHEADER bmiHeader;

int imWidth = 0, imHeight = 0;

int iters = 2;

int nd = 0;

double times[4][6][3][3];

for (int i = 1; i < 5; i++)

{

char\* cstr = inBMP(i);

BMPRead(rgb\_in, header, bmiHeader, cstr);

imWidth = bmiHeader.biWidth;

imHeight = bmiHeader.biHeight;

for (int threads = 1; threads <= 4; threads++)

{

omp\_set\_num\_threads(threads);

for (int alg = 0; alg < 6; alg++)

{

int ksize = 7;

for (int j = 0; j < 3; j++)

{

if (threads == 1)

{

if (alg == 0 || alg == 1 || alg == 2) {

times[nd][alg][j][0] = TestIter(Functions[alg], rgb\_in, imHeight, imWidth, ksize, iters, i, alg);

times[nd][alg][j][1] = times[nd][alg][j][0];

times[nd][alg][j][2] = times[nd][alg][j][0];

}

}

else

{

if (alg != 0 && alg != 1 && alg != 2)

{

times[nd][alg][j][threads - 2] = TestIter(Functions[alg], rgb\_in, imHeight, imWidth, ksize, iters, i, alg);

}

}

ksize += 4;

}

}

}

delete[] cstr;

nd++;

}

ofstream fout("output.txt");

fout.imbue(locale("Russian"));

for (int ND = 0; ND < 4; ND++)

{

switch (ND)

{

case 0:

cout << "\n----------1280\*720----------" << endl;

break;

case 1:

cout << "\n----------1920\*1200----------" << endl;

break;

case 2:

cout << "\n----------2560\*1440----------" << endl;

break;

case 3:

cout << "\n----------3840\*2160----------" << endl;

break;

default:

break;

}

for (int alg = 0; alg < 6; alg++)

{

for (int threads = 1; threads <= 4; threads++)

{

cout << "Thread " << threads << " --------------" << endl;

for (int j = 0; j < 3; j++)

{

cout << "Ksize = " << j << " --------------" << endl;

if (threads == 1)

{

if (alg == 0 || alg == 1 || alg == 2) {

cout << function\_names[alg] << "\t" << times[ND][alg][j][0] << " ms." << endl;

fout << times[ND][alg][j][0] << endl;

}

}

else

{

if (alg != 0 && alg != 1 && alg != 2)

{

cout << function\_names[alg] << "\t" << times[ND][alg][j][threads - 2] << " ms." << endl;

fout << times[ND][alg][j][threads - 2] << endl;

}

}

}

}

}

}

fout.close();

}

int main()

{

void\*\* FunctionsINT = new void\* [6] { testMedianLinearFilter, testMedianGaussFilter, testMedianQuickLinearFilter, testMedianLinearFilterParallel,

testMedianGaussFilterParallel, testMedianQuickLinearFilterParallel };

string function\_names[6]{ "Median Filter(Linear)", "Median Filter(Gauss)",

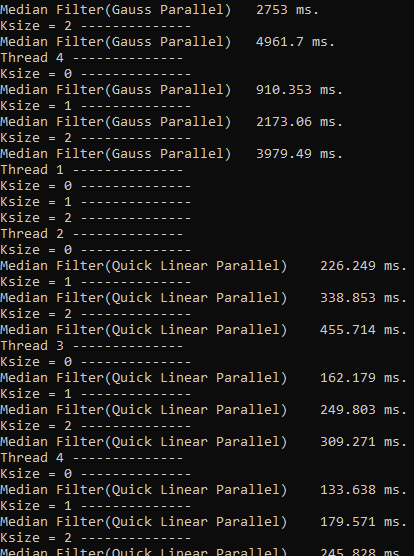
"Median Filter(Quick Linear)", "Median Filter(Linear Parallel)", "Median Filter(Gauss Parallel)","Median Filter(Quick Linear Parallel)" };

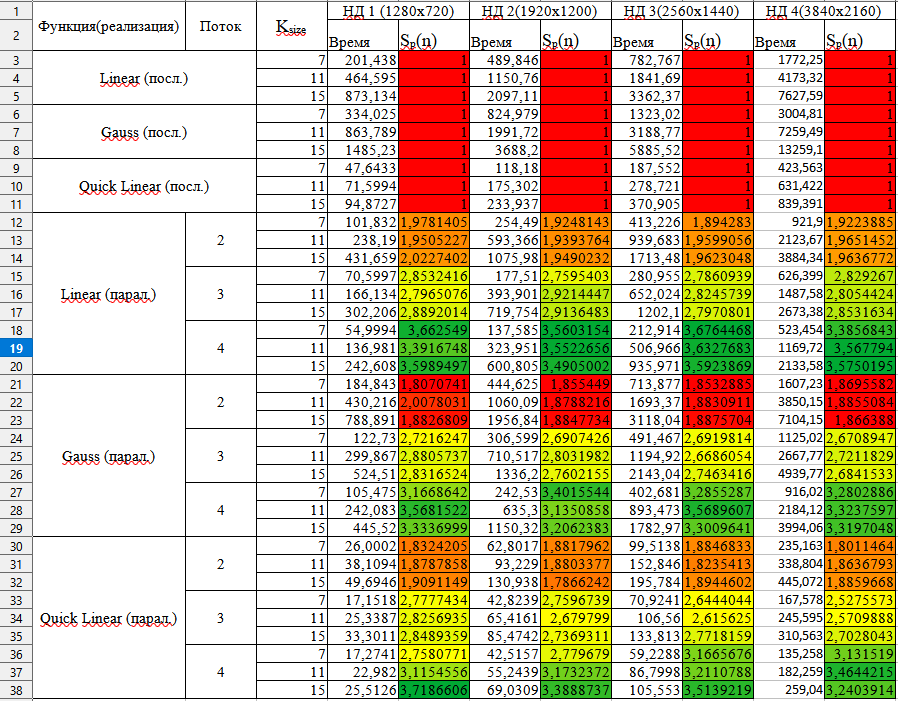
test\_functions(FunctionsINT, function\_names);

return 0;

}

Демонстрация работы программы представлена на рисунке 1.

Рисунок 1 – работа программы

Таблица 1 – результаты замеров времени работы первого задания с разными наборами функций

Вывод: в ходе выполнения работы были получены замеры времени выполнения работы функций в последовательном и многопоточном режимах. В задании работа функций в параллельном режиме с использованием алгоритма линейной фильтрации на 4 потоках имеет наибольшее ускорение, после него очень близок алгоритм фильтра Гаусса на 4 потоках, а самое наименьшее ускорение у алгоритма фильтра Гаусса на 2 потоках.

# ВЫВОДЫ

Были изучены различные алгоритмы выполнения линейной фильтрации на примере среднеарифметического фильтра и фильтра Гаусса с его различными аппроксимациями. Были получены навыки разработки параллельных алгоритмов, реализующих линейные фильтры, приняты во внимания подходы к оптимизации обработки.