Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского

Институт информационных технологий, математики и механики

**Отчет по лабораторной работе**

**«**Линейная фильтрация изображений (горизонтальное разбиение). Ядро Гаусса 3x3.**»**

**Выполнил**:

студент группы 381708-2

Соколюк Н. А.

**Проверил**:

Волокитин В.Д.

Нижний Новгород

2019

**Содержание**

[Введение 3](#_Toc33118001)

[Постановка задачи 4](#_Toc33118002)

[Метод решения 5](#_Toc33118003)

[Схема распараллеливания 7](#_Toc33118004)

[Описание программной реализации 7](#_Toc33118005)

[Результаты экспериментов 9](#_Toc33118006)

[Заключение 9](#_Toc33118007)

[Список литературы 10](#_Toc33118008)

[Приложение 11](#_Toc33118009)

# Введение

Линейная оконная фильтрация изображений в пространственной области заключается в вычислении линейной комбинации значений яркости пикселов в окне фильтрации с коэффициентами матрицы весов фильтра, называемой также маской или ядром линейного фильтра.

Рассмотрим вычисление такой линейной комбинации на примере окна фильтрации размером 3×33×3. При этом маска фильтра представляется матрицей вида:

|  |  |  |
| --- | --- | --- |
| Mask[−1,−1]Mask[−1,−1] | Mask[0,−1]Mask[0,−1] | Mask[1,−1]Mask[1,−1] |
| Mask[−1,0]Mask[−1,0] | Mask[0,0]Mask[0,0] | Mask[1,0]Mask[1,0] |
| Mask[−1,1]Mask[−1,1] | Mask[0,1]Mask[0,1] | Mask[1,1]Mask[1,1], |

а соответствующий фрагмент изображения с центральным пикселом Im[x,y][x,y], к которому на текущем шаге применяется данный фильтр, имеет вид:

|  |  |  |
| --- | --- | --- |
| Im[x−1,y−1]Im[x−1,y−1] | Im[x,y−1]Im[x,y−1] | Im[x+1,y−1]Im[x+1,y−1] |
| Im[x−1,y]Im[x−1,y] | Im[x,y]Im[x,y] | Im[x+1,y]Im[x+1,y] |
| Im[x−1,y+1]Im[x−1,y+1] | Im[x,y+1]Im[x,y+1] | Im[x+1,y+1]Im[x+1,y+1]. |

Результат линейной фильтрации для данного окна (для данного центрального пиксела) описывается следующей простой формулой:

Im′[x,y]=∑i=−hWinXhWinX ∑j=−hWinYhWinYIm[x+i, y+j]⋅Mask[x+i, y+j],(1)(1)Im′[x,y]=∑i=−hWinXhWinX ∑j=−hWinYhWinYIm[x+i, y+j]⋅Mask[x+i, y+j],

где hWinX=[WinX/2]hWinX=[WinX/2], hWinY=[WinY/2]hWinY=[WinY/2] - полуширина и полувысота окна фильтрации соответственно.

Результат применения операции ко всем пикселам изображения Im[x,y][x,y] называется сверткой изображения Im с маской Mask.

Ранее мы рассмотрели "вырожденный" случай линейной фильтрации с однородной маской. Между тем, сама идея свертки изображения с весовой матрицей аналогична ранее рассмотренной идее введения весовой матрицы во взвешенных процентильных фильтрах. Повысить устойчивость результатов фильтрации на краях областей можно, если придать более близким точкам окрестности большее влияние на окончательный результат, чем дальним.

Такая маска называется гауссовой; соответственно, и использующий ее линейный фильтр также называется *гауссовым*. Используя дискретные приближения двумерной гауссовой функции, можно получить и другие гауссовы ядра большего размера. Обратите внимание на то, что сглаживающие или фильтрующие маски линейных фильтров должны иметь сумму всех элементов, равную 11. Данное условие нормировки гарантирует адекватный отклик фильтра на постоянный сигнал (постоянное изображение Im[x,y]=\const[x,y]=\const).

# 

# Постановка задачи

1. Реализовать линейную фильтрацию изображений (горизонтальным разбиением). Ядро Гаусса 3x3.
2. Реализовать параллельный алгоритм линейной фильтрации изображений (горизонтальным разбиением). Ядро Гаусса 3x3.
3. Провести вычислительные эксперименты.
4. Сравнить время работы последовательного и параллельного алгоритмов.

# Метод решения

На вход поступает количество строк и столбцов матрицы. С помощью **std::mt19937** заполняем вектор структуры **struct Pixel**(содержащий 4 переменных типа **uint8\_t**) псевдорандомными случайными числами. Следующим шагом по циклам строк и столбцов, а также по циклу отношения средних значений отрицательного как меньше или равно положительному размеру ядра Гаусса, мы перебираем пиксели нашего вектора, через переменную **struct Pixel p** и записываем в переменные **r, g, b** элемент массива ядра Гаусса по индексу Гаусса, умноженный на значения переменных **r, g, b** переменной **p** структурой **Pixel**. После завершения цикла отношения средних значений отрицательного как меньше или равно положительному размеру ядра Гаусса, переменные **r, g, b**  мы делим на нормальное ядро Гаусса( то есть нулевое). И переменные **r, g, b** переменной **p** находим как максимальное между 0 и минимальным между найденными переменными **r, g, b** и 255(потому что значения любого пикселя в цветовом сегменте находится в диапазоне между 0 и 255.

# Схема распараллеливания

В начале находим целое количество строк, которое можно дать каждому процессу и остаток. Мы находим начальную строку равную целой части плюс остаток для первого процесса получателя, при условии, что строк более 2. Теперь количество строк мы приравниваем к начальной строке +1 для всех процессов. Находим начало **Begin**  как (начальная строка – 1)\* количество столбцов и отправляем каждому процессу.

# Описание программной реализации

В программе реализовано 7 функций:

# std::vector <Pixel> getRandomMatrix(int rows, int cols) - заполнение матрицы псевдослучайными значениями.

**std::vector <Pixel> solveSequential(std::vector <Pixel> a, int rows, int cols)** - последовательная линейная фильтрация изображений.

**Std::vector <Pixel> solveParallel(std::vector <Pixel> a, int rows, int cols)** - параллельная линейная фильтрация изображений.

**int main(int argc, char\* argv[]) -**  точка входа в программу.

# Результаты экспериментов

Эксперименты проводились на ПК с следующей конфигурацией:

* Операционная система: Windows 7 Максимальная
* Процессор: Intel(R) Core™ i5-3210M CPU @ 2.50 GHz
* ОЗУ 4 гб
* Версия Visual Studio: 2017

Размер массива 5000х5000 элементов.

Таблица 1. Результаты экспериментов

|  |  |  |  |
| --- | --- | --- | --- |
| Количество процессов | Время работы последовательного алгоритма, сек | Время работы параллельного алгоритма, сек | Ускорение |
| 1 | 1,999 | 2,003 | 0,998 |
| 2 | 2,006 | 1,601 | 1,253 |
| 3 | 1,997 | 1,510 | 1,323 |
| 4 | 2,008 | 1,319 | 1,523 |
| 5 | 1,999 | 1,312 | 1,524 |

По данным экспериментов видно, что использование параллельной линейной фильтрации изображений более эффективно, чем использование последовательного алгоритма линейной фильтрации изображений.

# Заключение

В ходе работы были реализованы два алгоритма со линейной фильтрации изображений:

последовательная линейная фильтрация изображений и параллельная линейная фильтрация изображений. Вычислительные эксперименты показали, что

последовательный алгоритм уступает в производительности параллельному алгоритму, так как работает на большем объеме данных. Этот факт делает параллельную линейную фильтрацию изображения более предпочтительной для использования в реальных вычислительных задачах.

# Список литературы

1. <http://wiki.technicalvision.ru/index.php/%D0%9B%D0%B8%D0%BD%D0%B5%D0%B9%D0%BD%D0%B0%D1%8F_%D1%84%D0%B8%D0%BB%D1%8C%D1%82%D1%80%D0%B0%D1%86%D0%B8%D1%8F_%D0%B2_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B9_%D0%BE%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D0%B8>

2. Хабр, сообщество IT-специалистов [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/261777/

# Приложение

#include "pch.h"

#include "mpi.h"

#include <stdio.h>

#include <iostream>

#include <random>

#include <vector>

#include <ctime>

#include <algorithm>

#include <utility>

#include <numeric>

#include <stdexcept>

#include <list>

using namespace std;

struct Pixel {

    uint8\_t r, g, b, a;

};

static int offset = 0;

static const int gaussianKernelSize = 3;

static int gaussianKernel[] = {

    1, 2, 1,

    2, 4, 2,

    1, 2, 1

};

static int gaussianKernelNorm = std::accumulate(

    std::begin(gaussianKernel), std::end(gaussianKernel), 0

);

std::vector <Pixel> getRandomMatrix(int rows, int cols) {

    std::mt19937 gen;

    gen.seed((unsigned)time(0) + ++offset);

    std::uniform\_int\_distribution<int> dis(0, 255);

    std::vector <Pixel> result(rows \* cols);

    for (int i = 0; i < rows \* cols; ++i) {

        result[i] = { static\_cast<uint8\_t>(dis(gen)), static\_cast<uint8\_t>(dis(gen)), static\_cast<uint8\_t>(dis(gen)) };

    }

    return result;

}

std::vector <Pixel> solveSequential(std::vector <Pixel> a, int rows, int cols) {

    std::vector <Pixel> oldA(a);

    if (rows \* cols != static\_cast<int>(a.size())) {

        throw std::runtime\_error("Matrix sizes does not match");

    }

    if (a.size() == 0) {

        throw std::runtime\_error("Image is empty");

    }

    for (int i = 0; i < rows; ++i) {

        for (int j = 0; j < cols; ++j) {

            double r = 0, g = 0, b = 0;

            int gaussianIndex = 0;

            for (int l = -gaussianKernelSize / 2; l <= gaussianKernelSize / 2; ++l) {

                for (int k = -gaussianKernelSize / 2; k <= gaussianKernelSize / 2; ++k) {

                    int idX = std::max(0, std::min(i + l, rows - 1));

                    int idY = std::max(0, std::min(j + k, cols - 1));

                    Pixel p = oldA[idX \* cols + idY];

                    r += gaussianKernel[gaussianIndex] \* p.r;

                    g += gaussianKernel[gaussianIndex] \* p.g;

                    b += gaussianKernel[gaussianIndex] \* p.b;

                    ++gaussianIndex;

                }

            }

            r /= gaussianKernelNorm;

            g /= gaussianKernelNorm;

            b /= gaussianKernelNorm;

            Pixel &p = a[i \* cols + j];

            p.r = std::max(0, std::min(static\_cast<int>(r), 255));

            p.g = std::max(0, std::min(static\_cast<int>(g), 255));

            p.b = std::max(0, std::min(static\_cast<int>(b), 255));

        }

    }

    return a;

}

std::vector <Pixel> solveParallel(std::vector <Pixel> a, int rows, int cols) {

    int code = 0;

    if (rows \* cols != static\_cast<int>(a.size())) {

        code = 1;

    }

    else if (a.size() == 0) {

        code = 2;

    }

    MPI\_Bcast(&code, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    if (code == 1) {

        throw std::runtime\_error("Matrix sizes does not match");

    }

    if (code == 2) {

        throw std::runtime\_error("Image is empty");

    }

    int size, rank;

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);

    size = std::min(size, rows);

    const int delta = rows / size;

    const int rem = rows % size;

    int total\_rows = rows;

    if (rank >= size) {

        total\_rows = rows = 0;

    }

    if (rows > 2) {

        if (rank == 0) {

            int rowStart = delta + ((rem > 0) ? 1 : 0);

            rows = rowStart + (rank + 1 != size ? 1 : 0);

            for (int i = 1; i < size; ++i) {

                int rowsCount = delta + ((i < rem) ? 1 : 0);

                if (rowsCount == 0) continue;

                int begin = (rowStart - 1) \* cols;

                int bufferSize = (rowsCount + 1 + (i + 1 != size ? 1 : 0)) \* cols;

                MPI\_Send(a.data() + begin, bufferSize, MPI\_INT, i, 1, MPI\_COMM\_WORLD);

                rowStart += rowsCount;

            }

        }

        else {

            int rowsCount = delta + ((rank < rem) ? 1 : 0);

            if (rowsCount > 0) {

                rows = rowsCount + 1 + (rank + 1 != size ? 1 : 0);

                int bufferSize = rows \* cols;

                a.resize(bufferSize);

                MPI\_Status status;

                MPI\_Recv(a.data(), bufferSize, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, &status);

            }

            else {

                total\_rows = rows = 0;

            }

        }

    }

    else if (rank > 0) {

        total\_rows = rows = 0;

    }

    MPI\_Barrier(MPI\_COMM\_WORLD);

    std::vector <Pixel> oldA(rows \* cols);

    for (int i = 0; i < rows \* cols; ++i) {

        oldA[i] = a[i];

    }

    for (int i = 0; i < rows; ++i) {

        for (int j = 0; j < cols; ++j) {

            double r = 0, g = 0, b = 0;

            int gaussianIndex = 0;

            for (int l = -gaussianKernelSize / 2; l <= gaussianKernelSize / 2; ++l) {

                for (int k = -gaussianKernelSize / 2; k <= gaussianKernelSize / 2; ++k) {

                    int idX = std::max(0, std::min(i + l, rows - 1));

                    int idY = std::max(0, std::min(j + k, cols - 1));

                    Pixel p = oldA[idX \* cols + idY];

                    r += gaussianKernel[gaussianIndex] \* p.r;

                    g += gaussianKernel[gaussianIndex] \* p.g;

                    b += gaussianKernel[gaussianIndex] \* p.b;

                    ++gaussianIndex;

                }

            }

            r /= gaussianKernelNorm;

            g /= gaussianKernelNorm;

            b /= gaussianKernelNorm;

            Pixel &p = a[i \* cols + j];

            p.r = std::max(0, std::min(static\_cast<int>(r), 255));

            p.g = std::max(0, std::min(static\_cast<int>(g), 255));

            p.b = std::max(0, std::min(static\_cast<int>(b), 255));

        }

    }

    if (total\_rows > 2) {

        if (rank != 0) {

            int rowsCount = delta + ((rank < rem) ? 1 : 0);

            if (rowsCount > 0) {

                MPI\_Send(a.data() + 1 \* cols, rowsCount \* cols, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

            }

        }

        else {

            int rowStart = delta + ((rem > 0) ? 1 : 0);

            for (int i = 1; i < size; ++i) {

                int rowsCount = delta + ((i < rem) ? 1 : 0);

                if (rowsCount == 0) continue;

                MPI\_Status status;

                MPI\_Recv(a.data() + rowStart \* cols, rowsCount \* cols, MPI\_INT, i, 2, MPI\_COMM\_WORLD, &status);

                rowStart += rowsCount;

            }

        }

    }

    return a;

}

int main(int argc, char\* argv[]) {

    double TimeStartWork, TimeEndWork, TimeStartWork2, TimeEndWork2;

    bool test = false;

    int ProcSize, ProcRank;

    int rows = 10;

    int cols = 5;

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &ProcSize);

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &ProcRank);

    std::vector<Pixel> a;

    rows = atoi(argv[1]);

    cols = atoi(argv[2]);

    if (ProcRank == 0) {

        a = getRandomMatrix(rows, cols);

    }

    TimeStartWork2 = MPI\_Wtime();//time

    std::vector <Pixel> answer = solveParallel(a, rows, cols);

    TimeEndWork2 = MPI\_Wtime();//time

    if (ProcRank == 0) {

        TimeStartWork = MPI\_Wtime();//time

        std::vector <Pixel> seqAnswer = solveSequential(a, rows, cols);

        TimeEndWork = MPI\_Wtime();//time

        for (int i = 0; i < rows; ++i) {

            for (int j = 0; j < cols; ++j) {

                Pixel p = seqAnswer[i \* cols + j];

                Pixel q = answer[i \* cols + j];

                if (p.r == q.r || p.g == q.g || p.b == q.b)

                    test = true;

                else {

                    return -1;

                    test = false;

                }

            }

        }

        if (ProcRank == 0) {

            if (test = true) {

                cout << "\nDate In Paralel = Date In Single"  << endl;

                cout << "\nTime In Solo = " << TimeEndWork - TimeStartWork << " sec" << endl;

                cout << "\nTime In Paralel = " << TimeEndWork2 - TimeStartWork2 << " sec" << endl;

            }

            else

                cout << "\nERRORE" << endl;

        }

    }

    MPI\_Finalize();

    return 0;

}