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 (блочное разбиение). Ядро Гаусса 3x3»

Выполнил:
студент группы 381706-2
Ясакова А. Е.

Проверил:
Доцент кафедры МОСТ,
Кандидат технических наук
Сысоев А. В.

Нижний Новгород
2019

Содержание

Введение

Под фильтрацией изображений понимают операцию, имеющую своим результатом изображение того же размера, полученное из исходного по некоторым правилам (фильтрам). Обычно цвет каждого пикселя результирующего изображения обусловлена цветами пикселей, расположенных в некоторой его окрестности в исходном изображении.

Методы линейной фильтрации предназначены для улучшения качества зашумленных изображений с сохранением перепадов яркости. Значительная часть алгоритмов фильтрации описывается локальным оператором свёртки.

Целью данной работы является реализация параллельной линейной фильтрации для изображения, заданного в оттенках серого, и сравнение ее производительности с последовательной реализацией.

Постановка задачи

В рамках лабораторной работы нужно реализовать параллельный алгоритм линейной фильтрации изображений (ядро гаусса 3x3).

Требуется выполнить следующее:

1. Последовательный алгоритм линейной фильтрации.
2. Параллельный алгоритм линейной фильтрации (блочное разбиение).
3. Провести вычислительные эксперименты.
4. Сравнить время работы параллельной и последовательной реализаций.

Метод решения

Пусть задано исходное изображение A, и обозначим интенсивности его пикселей A(x, y). Линейный фильтр определяется функцией F, заданной на растре. Данная функция называется ядром фильтра, а сама фильтрация производится при помощи операции дискретной свертки (взвешенного суммирования):

$$B(x, y) = \sum\sum F(i, j) * A(x + i, y + j)$$

Суммирование производится по (i, j), и значение каждого пикселя B(x, y) определяется пикселями изображения A, которые лежат в окне N, центрированном в точке (x, y) (будем обозначать это множество N(x, y)). Ядро фильтра, заданное на прямоугольной окрестности N, может рассматриваться как матрица m на n, где длины сторон являются нечетными числами. В данной задаче рассматривается ядро 3x3.

Если пиксель (x, y) находится в окрестности краев изображения. В этом случае A(x + i, y + j) в может соответствовать пикселю A, лежащему за пределами изображения A. Данную проблему можно разрешить несколькими способами:

1. Не проводить фильтрацию для таких пикселей, обрезав изображение B по краям или закрасив их, к примеру, черным цветом.
2. Не включать соответствующий пиксель в суммирование, распределив его вес F(i, j) равномерно среди других пикселей окрестности N(x, y).
3. Доопределить значения пикселей за границами изображения при помощи экстраполяции.
4. Доопределить значения пикселей за границами изображения, при помощи зеркального отражения.

В данной реализации линейной фильтрации был применен второй пункт.

Схема распараллеливания

В начале все процессы коммуникатора MPI_COMM_WORLD вычисляют свою часть матрицы (начальные строка и столбец, количество строк и столбцов).

Деление матрицы происходит следующим образом: делим количество столбцов и количество строк на корень из количества процессов, остатки от деления распределяем между процессами.

После деления матрицы каждый процесс (кроме нулевого) отправляет нулевому свои начальные строку и столбец, количество строк и столбцов. Это осуществляется с помощью MPI_Send и MPI_Resv. Нулевой процесс собирает все данные в один вектор для дальнейшего удобства вставки в результирующую матрицу.

Далее все процессы получают на вход матрицу целиком. Это осуществляется с помощью MPI_Bcast. Каждый процесс вычисляет свои значения для новой матрицы и отправляет нулевому процессу (MPI_Send и MPI_Resv). В конце нулевой процесс все полученные результаты складывает в нужные позиции результирующей матрицы.

В итоге результирующая матрица будет расположена в нулевом процессе.

Описание программной реализации

В программе реализованы методы:

- `std::vector <int> getImage(int rows, int cols)` – генерирование рандомной матрицы с соответствующими размерами (диапазон значений: от 0 до 255)
- `int CountPixel(const std::vector <int>& mask, int mask_size, int norm, const std::vector <int>& mtx, int i, int j, int rows, int cols)` – вычисление значения для конкретного пикселя результирующей матрицы
- `std::vector <int> ParallelLinearFilter(const std::vector <int> mask, std::vector <int> a, int rows, int cols)` – выполнение параллельной линейной фильтрации
- `std::vector <int> SequentialLinearFilter(const std::vector<int> mask, std::vector <int> a, int rows, int cols)` – выполнение последовательной линейной фильтрации

Подтверждение корректности

Для подтверждения корректности в программе реализован набор тестов с использованием библиотеки для модульного тестирования Google C++ Testing Framework. Тесты проверяют корректную работу последовательного и параллельного алгоритмов.

Дополнительно для подтверждения корректности в программе реализована демонстрация работы алгоритма визуально с использованием библиотеки OpenCV.

Результаты экспериментов

Характеристики ПК:

- Процессор: Intel® Core™ i5-85250U CPU @ 1.60GHz 1.80 GHz
- Оперативная память: 8,00 ГБ
- Операционная система: Windows 10

Количество процессов	Время выполнения последовательного алгоритма	Время работы параллельного алгоритма
2	4,74967	2,36442
3	4,74969	1,49473
4	4,7042	1,35564

Таблица 1. Результаты вычислений экспериментов

В рамках эксперимента было вычислено время работы последовательного параллельного алгоритмов линейной фильтрации. Размер изображения 3840x2160.

В ходе эксперимента было доказано, что алгоритм реализован эффективно, ускорение практически линейное. При увеличении количества процессов, ускорение с каждым разом уменьшается. Это объясняется увеличением накладных расходов.

Заключение

В ходе работы были реализованы последовательный и параллельный алгоритмы линейной фильтрации изображений (ядро Гаусса 3x3). Из результатов вычислительных экспериментов видим, что параллельный алгоритм эффективнее последовательного, так как последовательный алгоритм обрабатывает всю матрицу целиком.

Список литературы

Книги:

- В.П. Гергель, Р.Г. Стронгин. Основы параллельных вычислений для многопроцессорных вычислительных систем. Учебное пособие.

Интернет-ресурсы:

- Цифровая обработка сигналов. Обработка изображений. Фильтрация изображений. URL: https://studfile.net/preview/5830083/page:4/

Приложение

linear_filtering.h:

```
// Copyright 2019 Yasakova Anastasia
#ifndef MODULES_TASK_3_YASAKOVA_A_LINEAR_FILTERING_LINEAR_FILTERING_H_
#define MODULES_TASK_3_YASAKOVA_A_LINEAR_FILTERING_LINEAR_FILTERING_H_

#include <vector>


std::vector <int> getImage(int rows, int cols);
int CountPixel(const std::vector <int>& mask, int mask_size, int norm,
    const std::vector <int>& mtx, int i, int j, int rows, int cols);
std::vector <int> ParallelLinearFilter(const std::vector<int> mask,
std::vector <int> a, int rows, int cols);
std::vector <int> SequentialLinearFilter(const std::vector<int> mask,
std::vector <int> a, int rows, int cols);

#endif  // MODULES_TASK_3_YASAKOVA_A_LINEAR_FILTERING_LINEAR_FILTERING_H_
```

linear_filtering.cpp:

```
// Copyright 2019 Yasakova Anastasia
#include <mpi.h>
#include <vector>
#include <iostream>
#include <ctime>
#include <algorithm>
#include <cmath>
#include <numeric>
#include <random>
#include
"../../../modules/task_3/yasakova_a_linear_filtering/linear_filtering.h"



std::vector <int> getImage(int rows, int cols) {
    if (rows <= 0 || cols <= 0) {
        throw - 1;
    }
    std::vector <int> Matrix(rows * cols);
    std::default_random_engine random;
    random.seed(static_cast<unsigned int>(std::time(0)));
    for (int i = 0; i < rows * cols; i++) {
        Matrix[i] = random() % 256;
    }
    return Matrix;
}

int CountPixel(const std::vector <int>& mask, int mask_size, int norm,
    const std::vector <int>& mtx, int i, int j, int rows, int cols) {
    int p = 0;
    for (int l = -1; l < 2; ++l) {
        for (int k = -1; k < 2; ++k) {
            int i_ = i + l;
            int j_ = j + k;
            if (i_ * cols + j_ >= cols * rows) {
```

```
                i_ = i;
                j_ = j;
            }
            if (i_ < 0 || i_ > rows - 1) {
                i_ = i;
            }
            if (j_ < 0 || j_ > cols - 1) {
                j_ = j;
            }
            p += mtx[i_ * cols + j_] * mask[(l + 1) * 3 + k + 1];
        }
    }
    if (norm != 0) {
        p = p / norm;
    }
    if (p < 0) {
        p = 0;
    } else if (p > 255) {
        p = 255;
    }
    return p;
}

std::vector <int> ParallelLinearFilter(const std::vector<int> mask,
std::vector <int> mtx, int rows, int cols) {
    if (static_cast<int>(mtx.size()) != rows * cols) {
        throw - 1;
    }
    int size, rank;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    int mask_size = sqrt(mask.size());
    int norm = std::accumulate(mask.begin(), mask.end(), 0);
    std::vector<int> index(size * 4);
    int i_ = 0, j_ = 0;
    int rows_ = 0, cols_ = 0;
    int sqrt_size = floor(sqrt(size));
    float sqrt_size_rest = sqrt(size) - sqrt_size;
    int amount_cols = cols / sqrt_size, rest_cols = cols % sqrt_size;
    int amount_rows = 0, rest_rows = 0;
    if (sqrt_size_rest == 0) {
        amount_rows = rows / sqrt_size;
        rest_rows = rows % sqrt_size;
    } else if (fabs(sqrt_size - sqrt_size_rest) < 1) {
        amount_rows = rows - static_cast<int>((rows / (sqrt_size +
0.5)))* sqrt_size;
        rest_rows = static_cast<int>(rows / (sqrt_size + 0.5)) -
amount_rows;
    } else {
        amount_rows = rows / (sqrt_size + 1);
        rest_rows = rows % (sqrt_size + 1);
    }
    if (size < rows * cols) {
        MPI_Bcast(&mtx[0], rows * cols, MPI_INT, 0, MPI_COMM_WORLD);
        int different = size - sqrt_size * sqrt_size;
        if (different > 0 && rank >= sqrt_size * sqrt_size) {
            i_ = amount_rows * sqrt_size;
            sqrt_size = size - sqrt_size * sqrt_size;
            amount_cols = cols / sqrt_size;
```

```
            rest_cols = cols % sqrt_size;
            rows_ = amount_rows + rest_rows;
            if (rank % sqrt_size < rest_cols) {
                j_ = (amount_cols + 1) * (rank % sqrt_size);
                cols_ = amount_cols + 1;
            } else {
                j_ = amount_cols * (rank % sqrt_size) + rest_cols;
                cols_ = amount_cols;
            }
        } else {
            if (rank % sqrt_size < rest_cols) {
                j_ = (amount_cols + 1) * (rank % sqrt_size);
                cols_ = amount_cols + 1;
            } else {
                j_ = amount_cols * (rank % sqrt_size) + rest_cols;
                cols_ = amount_cols;
            }
            if (rank >= size - sqrt_size) {
                rows_ = amount_rows + rest_rows;
            } else {
                rows_ = amount_rows;
            }
            i_ = amount_rows * (rank / sqrt_size);
        }
        if (rank == 0) {
            MPI_Status status;
            index[0] = i_;
            index[1] = j_;
            index[2] = rows_;
            index[3] = cols_;
            for (int i = 1; i < size; i++) {
                MPI_Recv(&index[i * 4], 4, MPI_INT, i, 0, MPI_COMM_WORLD,
&status);
            }
        } else {
            std::vector<int>ind(4);
            ind[0] = i_;
            ind[1] = j_;
            ind[2] = rows_;
            ind[3] = cols_;
            MPI_Send(&ind[0], 4, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
    } else {
        i_ = 0;
        j_ = 0;
        rows_ = rows;
        cols_ = cols;
    }
    std::vector<int> res(rows * cols);
    if (rank == 0) {
        for (int i = i_; i < i_ + rows_; i++) {
            for (int j = j_; j < j_ + cols_; j++) {
                res[i * cols + j] = CountPixel(mask, mask_size, norm,
mtx, i, j, rows, cols);
            }
        }
        if (size < rows * cols) {
            MPI_Status status;
            for (int proc = 1; proc < size; proc++) {
```

```
                std::vector<int> res_(index[proc * 4 + 2] * index[proc *
4 + 3]);
                MPI_Recv(&res_[0], index[proc * 4 + 2] * index[proc * 4 +
3], MPI_INT, proc, 1, MPI_COMM_WORLD,
                    &status);
                int count = 0;
                for (int i = index[proc * 4]; i < index[proc * 4] +
index[proc * 4 + 2]; i++) {
                    for (int j = index[proc * 4 + 1]; j < index[proc * 4
+ 1] + index[proc * 4 + 3]; j++) {
                        res[i * cols + j] = res_[count];
                        count++;
                    }
                }
            }
        }
    } else {
        std::vector<int> res_;
        if (size < rows * cols) {
            for (int i = i_; i < i_ + rows_; i++) {
                for (int j = j_; j < j_ + cols_; j++) {
                    res_.push_back(CountPixel(mask, mask_size, norm, mtx,
i, j, rows, cols));
                }
            }
            MPI_Send(&res_[0], rows_ * cols_, MPI_INT, 0, 1,
MPI_COMM_WORLD);
        }
    }
    return res;
}


std::vector <int> SequentialLinearFilter(const std::vector<int> mask,
std::vector <int> mtx, int rows, int cols) {
    if (static_cast<int>(mtx.size()) != rows * cols) {
        throw - 1;
    }
    int mask_size = sqrt(mask.size());
    int norm = std::accumulate(mask.begin(), mask.end(), 0);
    std::vector <int> res;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            res.push_back(CountPixel(mask, mask_size, norm, mtx, i, j,
rows, cols));
        }
    }
    return res;
}
```

main.cpp:

```
// Copyright 2019 Yasakova Anastasia
#include <gtest-mpi-listener.hpp>
#include <gtest/gtest.h>
#include <vector>
#include <algorithm>
```

```
#include
"../../../modules/task_3/yasakova_a_linear_filtering/linear_filtering.h"


TEST(Linear_Filtering_MPI, test1) {
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        ASSERT_ANY_THROW(getImage(-1, 3));
    }
}

TEST(Linear_Filtering_MPI, test2) {
    std::vector <int> a;
    int rank;
    std::vector <int> mask = { 0, 0, 0, 0, 1, 0, 0, 0, 0 };
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        a = getImage(5, 7);
        ASSERT_ANY_THROW(ParallelLinearFilter(mask, a, 3, 2));
    }
}

TEST(Linear_Filtering_MPI, test3) {
    int rows = 4, cols = 5;
    std::vector <int> mask = { 1, 2, 1, 2, 4, 2, 1, 2, 1 };
    std::vector <int> a(rows * cols), ans_seq(rows * cols);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        a = { 150, 128, 100, 200, 175,
            100, 50, 250, 200, 255,
            0, 50, 75, 225, 130,
            145, 200, 30, 230, 100 };

        ans_seq = { 130, 123, 145, 183, 196,
            83, 98, 153, 196, 204,
            67, 85, 130, 175, 172,
            118, 132, 95, 172, 129 };
    }
    std::vector <int> ans(rows * cols);
    ans = ParallelLinearFilter(mask, a, rows, cols);
    if (rank == 0) {
        ASSERT_EQ(ans_seq, ans);
    }
}

TEST(Linear_Filtering_MPI, test4) {
    int rows = 40, cols = 80;
    std::vector <int> mask = { 1, 2, 1, 2, 4, 2, 1, 2, 1 };
    std::vector <int> a(rows * cols);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        a = getImage(rows, cols);
    }
    std::vector <int> ans(rows * cols);
    ans = ParallelLinearFilter(mask, a, rows, cols);
    std::vector <int> ans_seq(rows * cols);
```

```
    if (rank == 0) {
        ans_seq = SequentialLinearFilter(mask, a, rows, cols);
        ASSERT_EQ(ans_seq, ans);
    }
}

TEST(Linear_Filtering_MPI, test5) {
    int rows = 400, cols = 320;
    std::vector <int> mask = { 1, 2, 1, 2, 4, 2, 1, 2, 1 };
    std::vector <int> a(rows * cols);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    if (rank == 0) {
        a = getImage(rows, cols);
    }
    std::vector <int> ans(rows * cols);
    ans = ParallelLinearFilter(mask, a, rows, cols);
    std::vector <int> ans_seq(rows * cols);
    if (rank == 0) {
        ans_seq = SequentialLinearFilter(mask, a, rows, cols);
        ASSERT_EQ(ans_seq, ans);
    }
}

int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    MPI_Init(&argc, &argv);

    ::testing::AddGlobalTestEnvironment(new
GTestMPIListener::MPIEnvironment);
    ::testing::TestEventListeners& listeners =
        ::testing::UnitTest::GetInstance()->listeners();

    listeners.Release(listeners.default_result_printer());
    listeners.Release(listeners.default_xml_generator());

    listeners.Append(new GTestMPIListener::MPIMinimalistPrinter);
    return RUN_ALL_TESTS();
}
```