Автономное образовательное учреждение высшего профессионального образования
«Национальный исследовательский
Нижегородский государственный университет
им. Н.И. Лобачевского»

Институт информационных технологий математики и механики

Отчёт по лабораторной работе

Линейная фильтрация изображений (блочное разбиение). Ядро Гаусса 3x3

Выполнил:
Студент ИИТММ гр. 381608
Ермаченко Б. А.

Проверил:
Доцент кафедры МОСТ
Сысоев А. В.

Нижний Новгород
2019 г.

Оглавление

Введение

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

Правила фильтрации могут быть самыми разнообразными. Фильтрация изображений является одной из самых фундаментальных операций компьютерного зрения, распознавания образов и обработки изображений. С той или иной фильтрации исходных изображений начинается работа подавляющего большинства методов обработки изображений.

Линейные фильтры имеют очень простое математическое описание. Линейный фильтр определяется вещественнозначной функцией h (ядром фильтра), заданной на растре. Сама фильтрация производится при помощи операции дискретной свертки

Матрица свёртки – это матрица коэффициентов, которая «умножается» на значение пикселей изображения для получения требуемого результата.

Фильтр Гаусса — фильтр размытия изображения, который использует нормальное распределение (также называемое Гауссовым распределением) для вычисления преобразования, применяемого к каждому пикселю изображения.

Постановка задачи

Необходимо реализовать программу, которая фильтрует произвольное изображение с помощью фильтра Гаусса, матрица 3x3. Использовать блочное разбиение при параллельных вычислений

Цель данной работы:

1. Реализовать последовательную версию программы;
2. Реализовать OMP параллельную версию программы, используя блочное разбиение изображения.
3. Реализовать TBB параллельную версию программы, используя блочное разбиение изображения.
4. Оценить эффективность и масштабируемость данной программы на кластере (или многопроцессорный или многопоточный запуск на персональном компьютере).

Метод решения

Последовательная версия

Алгоритм работы последовательной версии программы делится на следующие части:

1. Генерация данных/Загрузка изображения
2. Фильтрация.
3. Вывод результатов/изображения.

Этап 1 реализован в двух версиях. В одной генерируются входные данные программы в виде матрицы положительных целых чисел, что имитирует пиксели изображения. Во второй при помощи возможностей OpenCV загружается реальное изображение

Этап 2 является основным этапом программы. В ходе него происходит фильтрация входного изображения при помощи матрицы свертки фильтра Гаусса. Размер матрицы можно задать. Чем больше размер матрицы, тем сильнее размывается изображение. Матрица высчитывается по специальной формуле.

Алгоритм фильтрации для фильтра Гаусса:

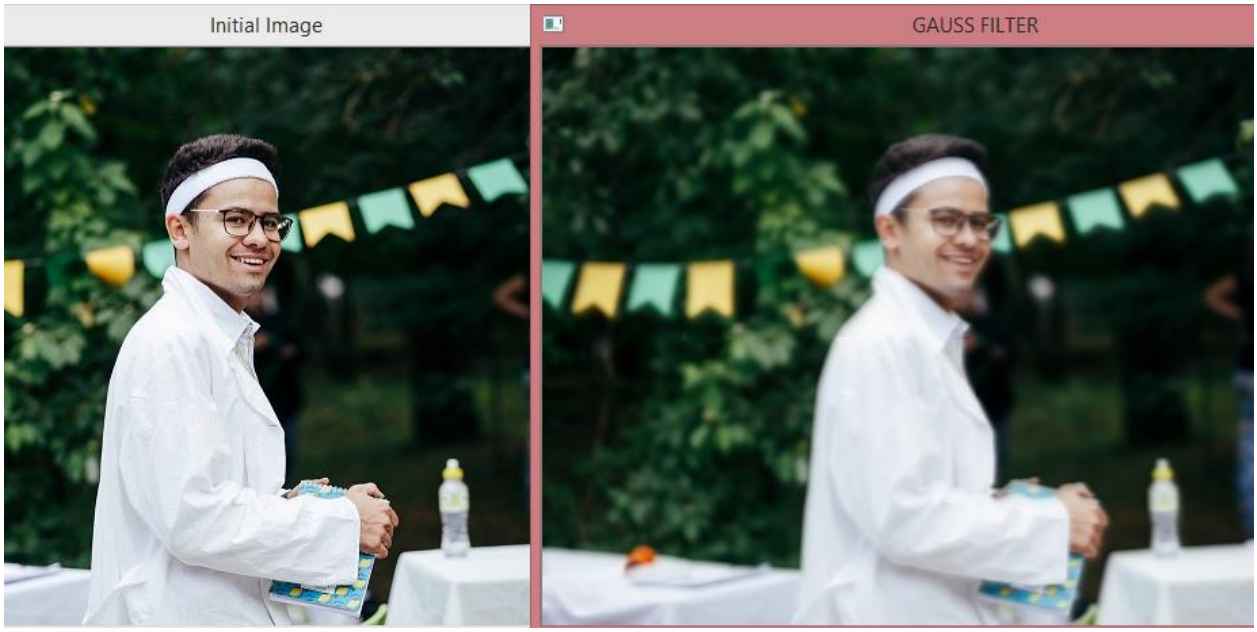
Ядро свёртки позволяет усилить или ослабить компоненты изображения. Матрица свертки проходит по каждому пикселю изображения, при этом матрица свертки в процессе остаётся неизменной . В каждой точке матрица Гаусса поэлементно умножается на значение соответствующих пикселей исходного изображения и произведения суммируются. Полученная сумма присваивается тому пикселю нового изображения, который соответствует положению центра матрицы. Результат записывается во временную матрицу, чтобы исключить влияние обработанных пикселей на необработанные. У крайних пикселей изображения всегда отсутствуют некоторые соседние пиксели, следовательно, нет данных для полных вычислений. Поэтому в программе пиксели на границах не обрабатываются. Заметим, что значения пикселей должны быть в диапазоне от 0 до 255, поэтому, если значение оказалось вне интервала, то оно соответственно приводится к 0 или 255.

Заметим, что этот алгоритм справедлив для всех реализаций - в том числе параллельных программ.

Алгоритм фильтрации для фильтра Гаусса:

```
for (int yi = 0; yi < height; yi++) {
    for (int xj = 0; xj < width; xj++) {
        double color = 0;
        double kSum = 0;
        for (int i = -n / 2; i < n / 2; i++) {
            for (int j = -n / 2; j < n / 2; j++) {
                kSum += w[i + n / 2][j + n / 2];
                if ((i + yi) >= 0 && (i + yi) < height
                    && (j + xj) >= 0 && (j + xj) < width)
                    color += arrImage[static_cast<int>(i + yi)][static_cast<int>(j + xj)]
                    * w[i + n / 2][j + n / 2];
            }
        }
        if (kSum <= 0) kSum = 1;
        color /= kSum;
        if (color < 0) color = 0;
        if (color > 255) color = 255;
        if ((yi) >= 0 && (yi)<height && (xj) >= 0 && (xj)<width)
            new_arrImage[static_cast<int>(yi)][static_cast<int>(xj)] = color;
    }
}
```

Пример результата работы алгоритма фильтра Гаусса, матрица 5x5:



Параллельная версия OpenMp

Алгоритм работы параллельной версии программы делится на следующие части:

1. Генерация данных/Загрузка изображения
2. Определение количества блоков.
3. Определение размера блоков.
4. Параллельная фильтрация блоков.
5. Вывод результатов/изображения

Этапы 1, 5 аналогичны этапам 1, 3 последовательной версии.

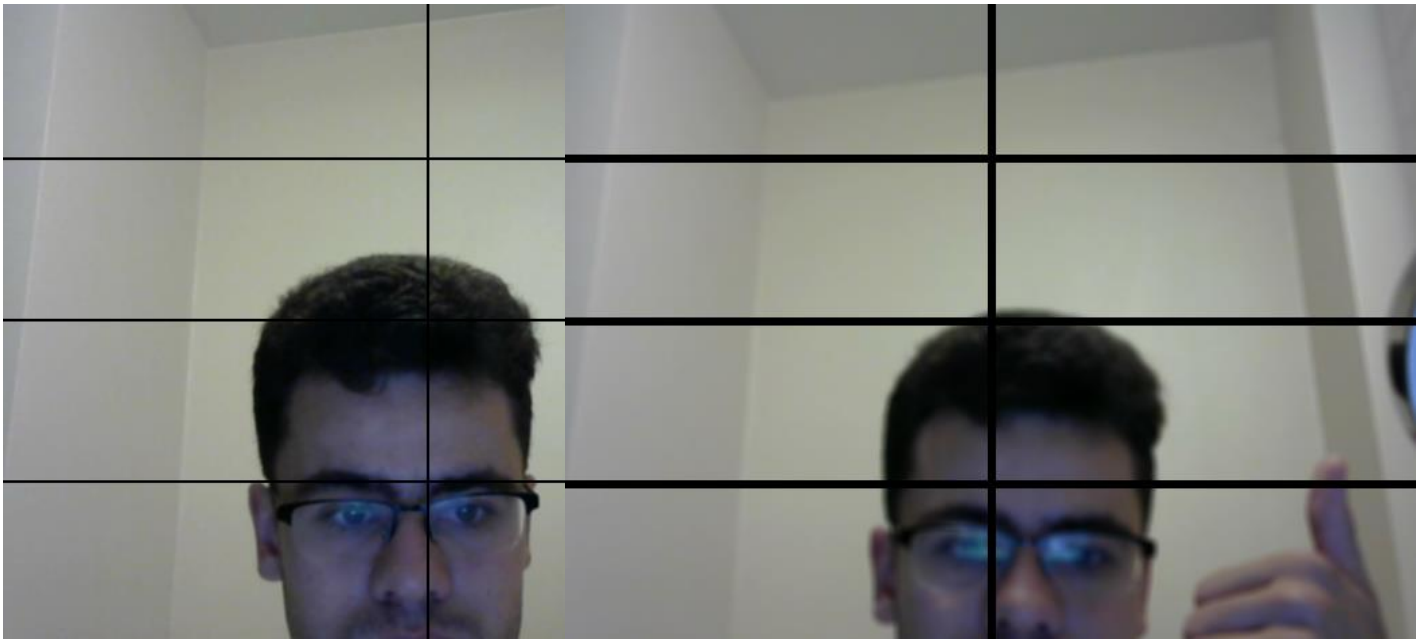
Этап 2 параллельной версии необходим для определения оптимального количества блоков и распределения их на расположение по высоте/ширине изображения. То есть сколько блоков будет по ширине, сколько по высоте. Количество блоков определяется по количеству потоков, которые задает пользователь

Этап 3 необходим для того чтобы узнать ширину и высоту каждого блока отталкиваясь от размеров изображения

Этап 4 параллельной версии отличается тем, что мы фильтруем только часть(блок) изображения. Для создания параллельной части используем директиву `#pragma omp parallel`. Чтобы найти расположения нашего блока, мы получаем номер потока и высчитываем место фрагмента в изображении, потом идет уже знакомая нам фильтрация фильтром Гаусса .

Корректность программы проверяется как полное совпадение выходных данных последовательной и параллельных версий на одном и том же наборе данных(при задании матрицы чисел), а также визуально при помощи OpenCV

Демонстрация работы на 8 потоках, матрица 5X5 с помощью веб камеры и библиотеки OpenCV



Этап 2. Алгоритм определения количества блоков по ширине/высоте:

```
int tmp = sqrt(throwds);
while (throwds % tmp != 0) {
    tmp--;
}
blockX = tmp;
       blockY = throwds / tmp;
```

Комментарий: находим корень из числа потоков, далее ищем ближайший к нему делитель исходного числа

Этап 2. Алгоритм определения размера блоков:

```
int* rezArrX = new int[blockX];
int rest = width % blockX;
for (int i = 0; i < blockX; i++) {
    rezArrX[i] = width / blockX;
}
if (rest != 0) {
    int k = 0;
    while (rest != 0) {
        rezArrX[k%rest]++;
        rest--;
        k++;
    }
}
       return rezArrX;
```

Комментарий: создаем массив для каждого фрагмента. делим ширину на количество блоков, присваиваем каждому члену массива. Если есть остаток - распределяем.

Этап 3.

Нахождение нужного фрагмента для фильтрации в картинке по номеру потока:

```
int numberThroat = omp_get_thread_num();
int startX = 0;
int startY = 0;
int row = 0;
int column = 0;
int count = 0;
for (int i = 0; i < blockY; i++) {
    if (i > 0) startY += lengthY[i-1];
    for (int j = 0; j < blockX; j++) {
        if (j > 0) startX += lengthX[j-1];
        count++;
        if (count > numberThroat)break;
        column++;
    }
    if (count > numberThroat)break;
    row++;
    column = 0;
    startX = 0;
       }
```

Комментарий: прибавляем длины отрезков по длине/по ширине соответственно, пока не дойдем до нужного блока.

Этапы 1, 5 аналогичны последовательной версии

Параллельная версия TBB

Алгоритм работы параллельной версии программы делится на следующие части:

1. Генерация данных/Загрузка изображения
2. Определение количества блоков.
3. Определение размера блоков.
4. Параллельная фильтрация блоков.
5. Вывод результатов/изображения

Логика TBB и OpenMP версий программ похожи.
В TBB-версии программы мы используем для распараллеливания двумерное итерационное пространство tbb::blocked_range2d, куда и передаем размеры блоков. При фильтрации нам не нужно высчитывать расположение фрагмента, так как tbb делает это за нас, предоставляя r.begin() и r.end().

Таким образом, алгоритм параллельной части будет выглядеть следующим образом:

```
parallel_for(blocked_range2d<int, int>(0, width, lengthX[0], 0, height, lengthY[0]),
[&](const blocked_range2d<int, int>& r)
    {
              for (int yi = r.cols().begin() +n/2; yi < r.cols().end()- n / 2; yi++) {
                  for (int xj = r.rows().begin() + n / 2; xj < r.rows().end() -n / 2;
xj++) {
                        double color = 0;
                        double kSum = 0;
                        double colorR = 0;
                        double colorB = 0;
                        double colorG = 0;
                        for (int i = -n / 2; i <= n / 2; i++) {
                            for (int j = -n / 2; j <= n / 2; j++) {
                                 if ((i + yi) >= 0 && (i + yi) < height
                                  && (j + xj) >= 0 && (j + xj) < width) {
                                kSum += w[i + n / 2][j + n / 2];
                                colorR += r1[i + yi][j + xj] * w[i + n / 2][j + n / 2];
                                colorG += g[i + yi][j + xj] * w[i + n / 2][j + n / 2];
                                colorB += b[i + yi][j + xj] * w[i + n / 2][j + n / 2];

                                }
                            }
                        }
                        if ((yi) >= 0 && (yi) < height &&
                             (xj) >= 0 && (xj) < width) {
                        if (kSum <= 0) kSum = 1;
                        colorR /= kSum;
                        if (colorR < 0) colorR = 0;
                        if (colorR > 255) colorR = 255;
                        rr[yi][xj] = colorR;
                        colorG /= kSum;
                        if (colorG < 0) colorG = 0;
                        if (colorG > 255) colorG = 255;
                        gr[yi][xj] = colorG;
                        colorB /= kSum;
                        if (colorB < 0) colorB = 0;
                        if (colorB > 255) colorB = 255;
                        br[yi][xj] = colorB;

                         }
                    }
                }

    });
```

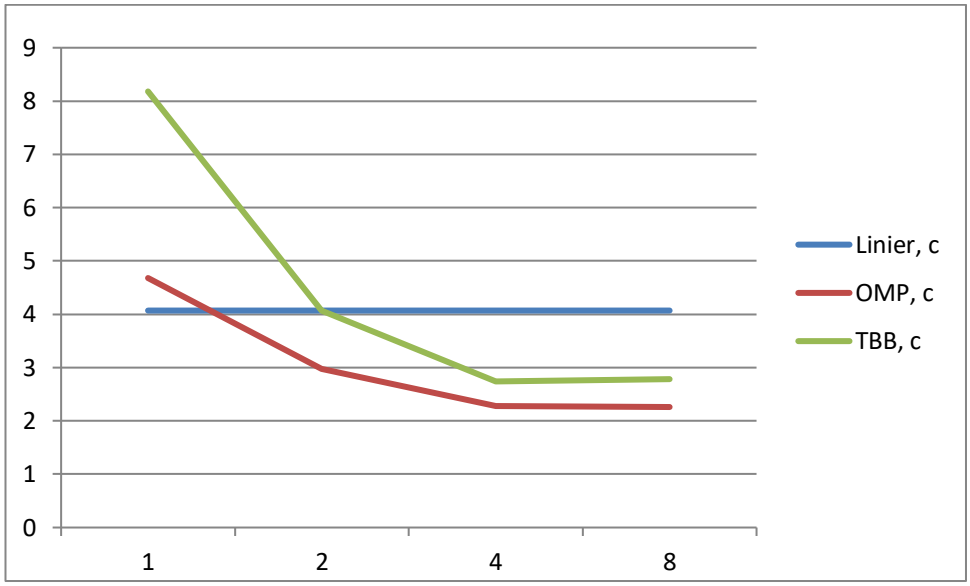
Результаты экспериментов

Эксперименты проводились на персональном компьютере с использованием многопроцессорного и многопоточного запусков.

Характеристики персонального компьютера:

1. Процессор Intel Core i5 CPU 2.50GHz
2. Оперативная память 6GB

Входная матрица 8000X8000 элементов целочисленного типа (int).



Ось ординат - время. По оси абсцисс - потоки.

Можем заметить, что при 1 потоке, последовательная версия работает быстрее, чем параллельные, так как TBB и OpenMP затрачивают время на накладные расходы, при этом также работают последовательно. При увеличении количества потоков, параллельные версии начинают работать быстрее последовательной, так как вычисления распараллеливаются по потокам. Также можем заметить, что при числе потоков больше 4 значительного ускорения не происходит, так как компьютер 4х-ядерный и увеличивать количество потоков более 4х практически не имеет смысла. Более того, программа может работать медленней за счет увеличения накладных расходов на распараллеливание.

Также заметим, что при количестве потоков равным 4, ускорение не происходит в 4 раза. Это происходит из-за того, что не вся программа распараллеливается, а только ее часть. То есть программа содержит значительную последовательную часть, например, генерацию матрицы. Кроме того дополнительное время уходит на операции распараллеливания, накладные расходы.

Если сравнивать OMP и TBB версии - получили, что TBB версия работает заметно медленней. Данные результаты можно пояснить тем, что в tbb мы используем двумерное пространство tbb::blocked_range2d, так как нам необходимо блочное разбиение. Таким образом увеличивается время на накладные расходы. Также в официальной статье говорится о том, что OpenMP, как правило, лучше работает с массивами данных https://software.intel.com/en-us/intel-threading-building-blocks-openmp-or-native-threads. Кроме того, возможно нужно учитывать особенности работы данного компьютера.

Заключение

В ходе работы был реализован последовательный и параллельный алгоритм фильтрации Фильтром Гаусса, блочное разбиение. Был реализован параллельный алгоритм для OpenMP и TBB версий. Была проведена серия экспериментов, которая позволяет оценить эффективность использования каждой из технологий и их совокупности.

Из результатов эксперимента видно, что TBB и OpenMP дают максимальное ускорение близкое к 2м, OpenMP работает быстрее. Обе эти технологии предназначены для решения задач в системах с общей памятью. OpenMP - встроенный стандарт в язык, TBB - отдельная библиотека

Источники

http://www.apmath.spbu.ru/ru/staff/pogozhev/lection07.pdf