САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе №5

«OpenMP»

Выполнил: Панюхин Никита Константинович

Номер ИСУ: 334964

студ. гр. М3138

Санкт-Петербург

2022

Цель работы: знакомство со стандартом OpenMP.

Инструментарий и требования к работе: рекомендуется использовать C, C++. Возможно использовать Python и Java. Стандарт OpenMP 2.0.

Теоретическая часть

Автоматическая контрастность изображений

Рассмотрим алгоритм автоматической настройки контрастности, используемый в данном проекте. На вход алгоритму подаётся изображение в виде массива цветных или чёрно-белых пикселей, значение каждого цвета (или, в ч/б варианте, целиком пикселя) лежит в диапазоне $[0; 255]$. На выходе ожидается изображение в таком же формате.

Автоматическая настройка контрастности подразумевает растяжение диапазона значений цветов. Например, если исходное изображение имело цвета от 20 до 230, то итоговое будет иметь полный диапазон, то есть $[0; 255]$. Растяжение диапазона цвета производится по следующей формуле:

$$new_{color} = (old_{color} - min_{color}) \cdot \frac{255}{max_{color} - min_{color}}$$

где old_{color} и new_{color} – старый и новый цвет пикселя соответственно, min_{color} и max_{color} – минимальный и максимальный найденные цвета в исходном изображении соответственно (в примере выше – 20 и 230).

Однако помимо исходного изображения на вход алгоритму также поступает коэффициент $coeff = [0; 0.5)$, обозначающий часть пикселей, которую при подсчёте следует проигнорировать с обоих концов. Например, если коэффициент равен 0.1, необходимо пропустить 10% пикселей с начала и конца диапазона $[20; 230]$, получив, условно, $[25; 220]$. Интуитивной реализацией было бы сложить все пиксели в один массив, отсортировать, отрезать границы и применить

растяжение, однако это работает слишком долго, за $O(n \cdot \log n)$. Далее будет показана быстрая реализация данного алгоритма.

Заметим, для того чтобы посчитать границы с учётом $coeff$ проигнорированных пикселей не нужно считать целый массив с сортировкой, поскольку каждый пиксель может иметь всего 256 различных значений. Таким образом, посчитав кол-во пикселей каждого из 256 цветов мы можем, с помощью префиксной суммы, найти нужные границы. Асимптотика: $O(n + 256 \cdot O(1)) = O(n)$. Производить подсчёт пикселей каждого из 256 цветов ($frequency$) возможно параллельно. Для этого в каждом потоке создаётся временный массив tmp_freq, в который читается ответ для выделенной данному потоку части изображения, и который в конце прибавляется к итоговому массиву $freq$. Более подробно эта оптимизация описана в части OpenMP.

После получения массива $frequency$, подсчёта на нём префиксной суммы и нахождения итоговых границ диапазона ($[25; 220]$ в примере), остаётся лишь применить указанную выше формулу к каждому пикселю. Снова заметим, что возможных значений пикселя всего 256 и посчитаем всевозможные переходы цветов заранее (массив $mapping[256]$). Тогда всё, что останется сделать – это применить $mapping$ для каждого пикселя изображения, что можно сделать одним циклом `for` в параллельном режиме.

Для бóльшей оптимизации кода заметим, что чёрно-белые и цветные изображения не отличаются в алгоритме. Более того, в алгоритме, по сути, нигде не использовались ширина, высота и размерность пикселей изображения. Это суждение позволяет рассматривать картинки, как большой непрерывный массив значений $[0; 255]$ (`uint8_t` в C++) без разбиения его на строки или пиксели. В коде можно сделать меньше циклов и проверок; большие и короткие циклы легче распараллеливаются.

OpenMP

OpenMP – открытый стандарт для распараллеливания программ на языках C, C++ и Fortran. Несмотря на то, что OpenMP имеет множество возможностей, которые применимы только к новым версиям, конкретно для данного проекта использовалась версия OpenMP 2.0. Далее будут вкратце рассмотрены основные аспекты работы с OpenMP, его возможные случаи и способы применения. Основой примеров будет служить выбранный для данного проекта язык C++.

Интерфейс взаимодействия с OpenMP заключается в использовании специального синтаксиса языка (прагм). Например, блок параллельности с 4 потоками в коде будет выглядеть так: `#pragma omp parallel num_threads(4)`. То есть, чтобы показать компилятору, что в данном месте необходима параллельность, нужно написать `parallel`. `#pragma omp` в свою очередь является обращением к модулю OpenMP. Для проекта также использовалось ключевое слово `for`, которое позволяет эффективно распараллелить циклы `for`. Ключевое слово `num_threads` позволяет установить желаемое количество потоков на блок параллельности. Того же эффекта можно добиться с помощью команды `omp_set_num_threads(n)`. Ключевое слово `critical` позволяет показать компилятору, какая часть кода должна выполняться атомарно, то есть одновременно только одним потоком для исключения коллизий.

В данном алгоритмы используются только инструкции $parallel$, $critical$ и for (опционально $section(s)$). Другие инструкции OpenMP, такие как $default$, $shared$, $private$ и т.п., несмотря на то, что были тщательно изучены, в процессе работы не понадобились из-за качественно продуманной структуры кода. Так, например, $shared$ переменные можно объявлять вне блока параллельности, а $private$ наоборот, внутри.

Для оптимизации простых циклов использовался модификатор `omp parallel for`. Для разделения боле сложных конструкций была применена следующая методика (для примера рассмотрим параллельный подсчёт пикселей, значения $[0; 255]$): инициализировался выходной ответ в виде массива и блок параллельности; затем в

каждом потоке создавался отдельный массив, идентичный главному ответу, а также, исходя из номера текущего потока и суммарного кол-ва потоков, высчитывался блок исходного изображения, который данный поток будет обрабатывать. После обработки потоком своего блока он относительно быстро записывал результат в итоговый массив. Таким образом исключается возможности коллизии и ошибки при записи в небольшой по размеру массив, при сохранении достаточно высокой скорости выполнения, в отличии от одного цикла с атомарной операцией увеличения значения на 1. Следует заметить, что можно улучшить данный алгоритм просто добавив ещё один или несколько слоёв такой параллельной обработки набора данных во временный массив и слияние временных массивов. Если размер массива не делится нацело на количество потоков, в конце нужно обработать немного оставшихся элементов.

Практическая часть

Для тестирования программы и обработки результатов была выбрана фотография, полученная путём наложения дополнительных слоёв (в том числе уменьшающих контрастность) на оригинальное изображение (см. Рисунки 1 и 2).

Рисунок 1 – Исходное изображение (637Мб), низкая контрастность

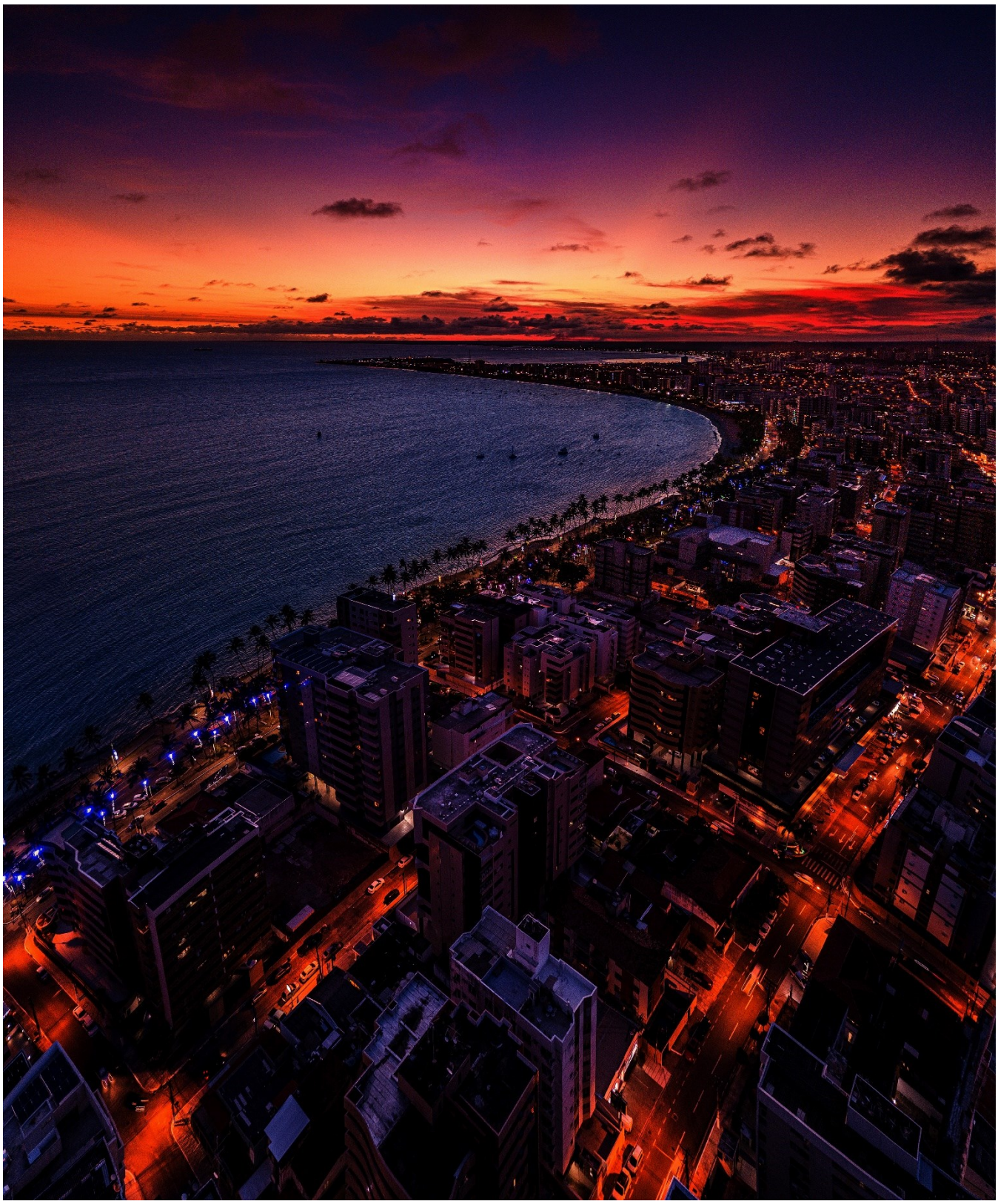

Рисунок 2 – Ожидаемый результат работы программы, высокая контрастность

Тестирование проводилось на компиляторе GCC со следующими настройками:

```
#pragma GCC optimize("Ofast")
#pragma GCC target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,mmx,tune=native")
#pragma GCC target("avx2")
```

Не смотря на возможные погрешности вычислений, после анализа результатов на `O3` был выбран уровень оптимизации `Ofast`. Никаких различий между ними обнаружено не было. Действительно – сложных или требующих большой точности вычислений в алгоритме нет.

Также в этом отчёте только одно тестовое изображение, поскольку в процессе изучения была установлена схожесть результатов различных картинок – показывать несколько неинформативно. В репозитории проекта присутствуют и другие изображения.

Далее представлены графики времени работы программы, зависящие от различных переменных величин, требуемых в задании (см. Рисунки 3, 4 и 5):

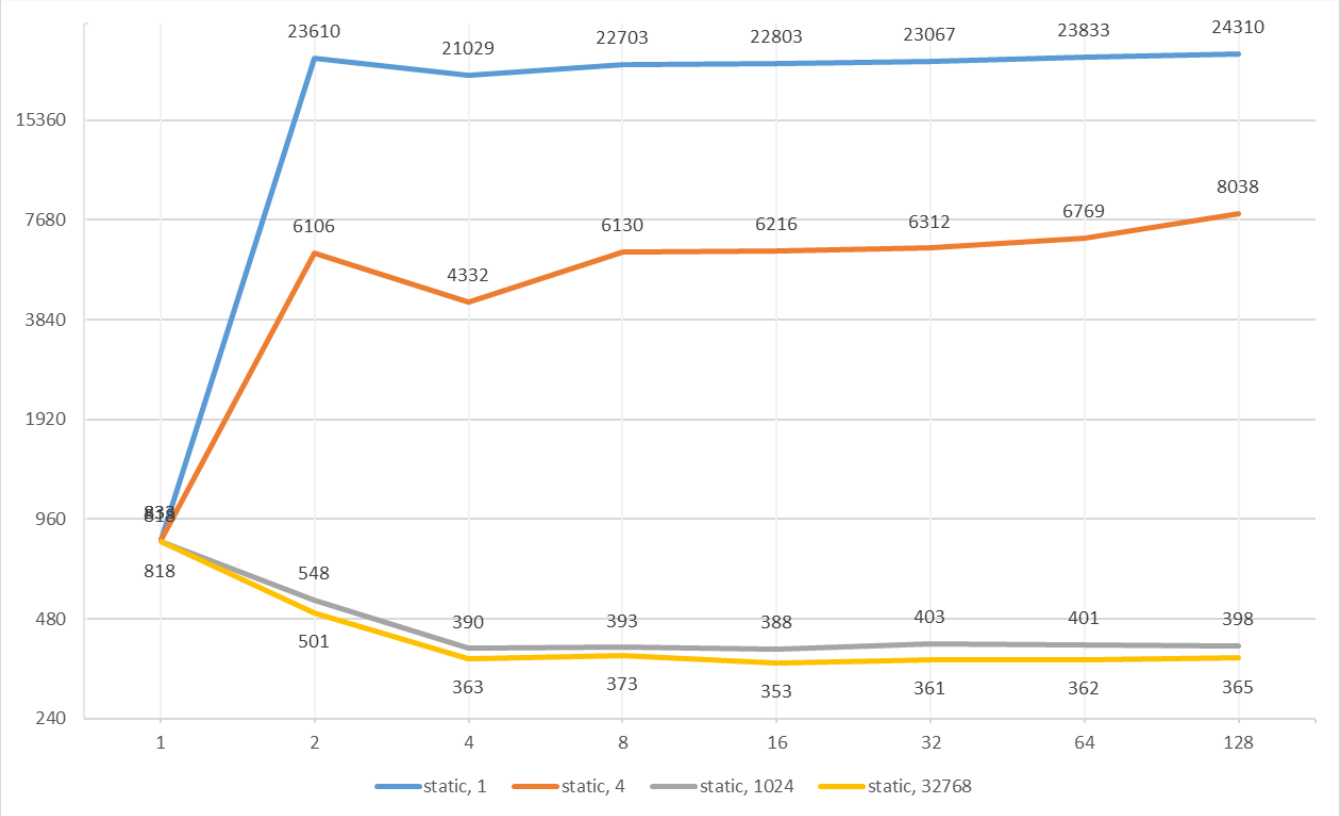

Рисунок 3 – зависимость времени работы от количества потоков при параметре $schedule = static$ и различными параметрами $chunk\_size$

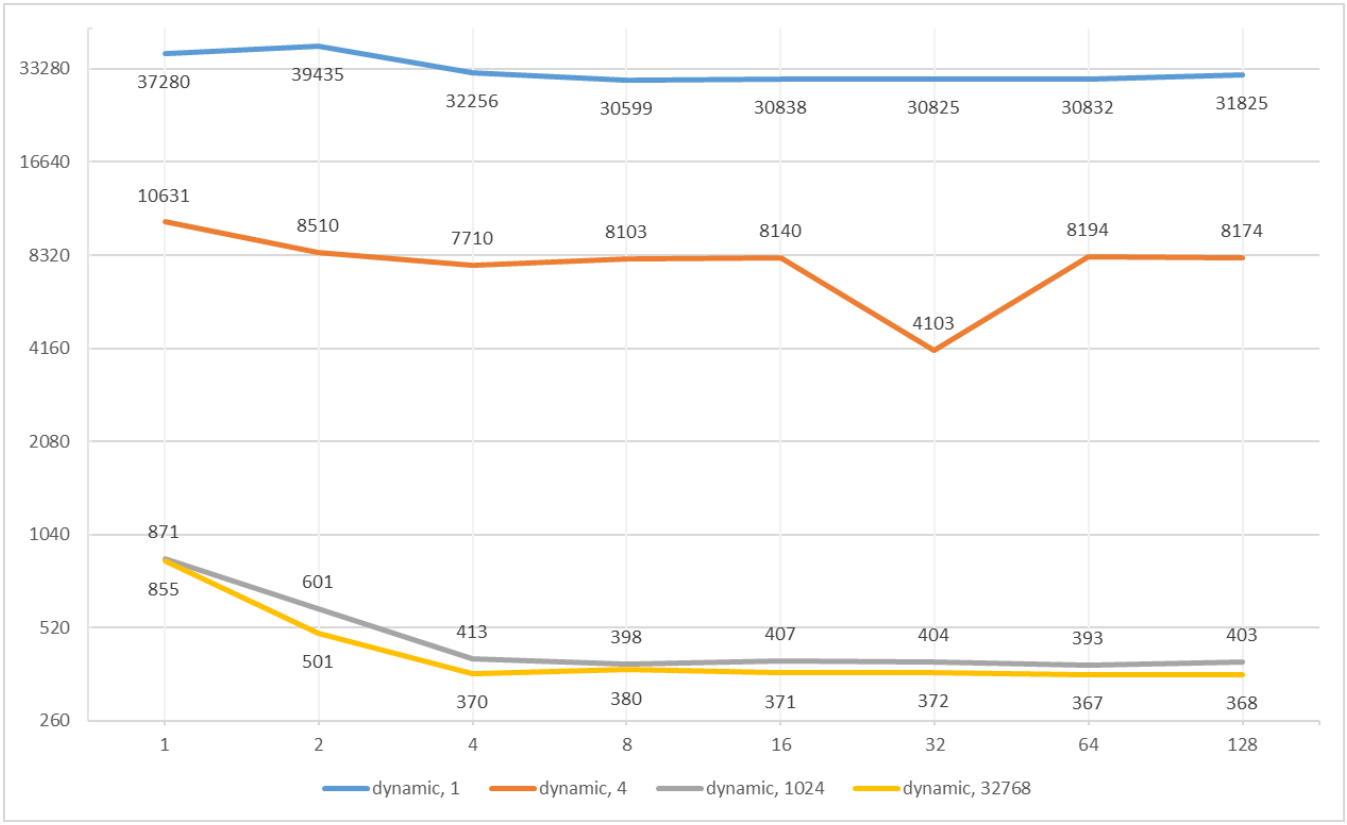

Рисунок 4 – зависимость времени работы от количества потоков при параметре $schedule = dynamic$ и различными параметрами $chunk\_size$

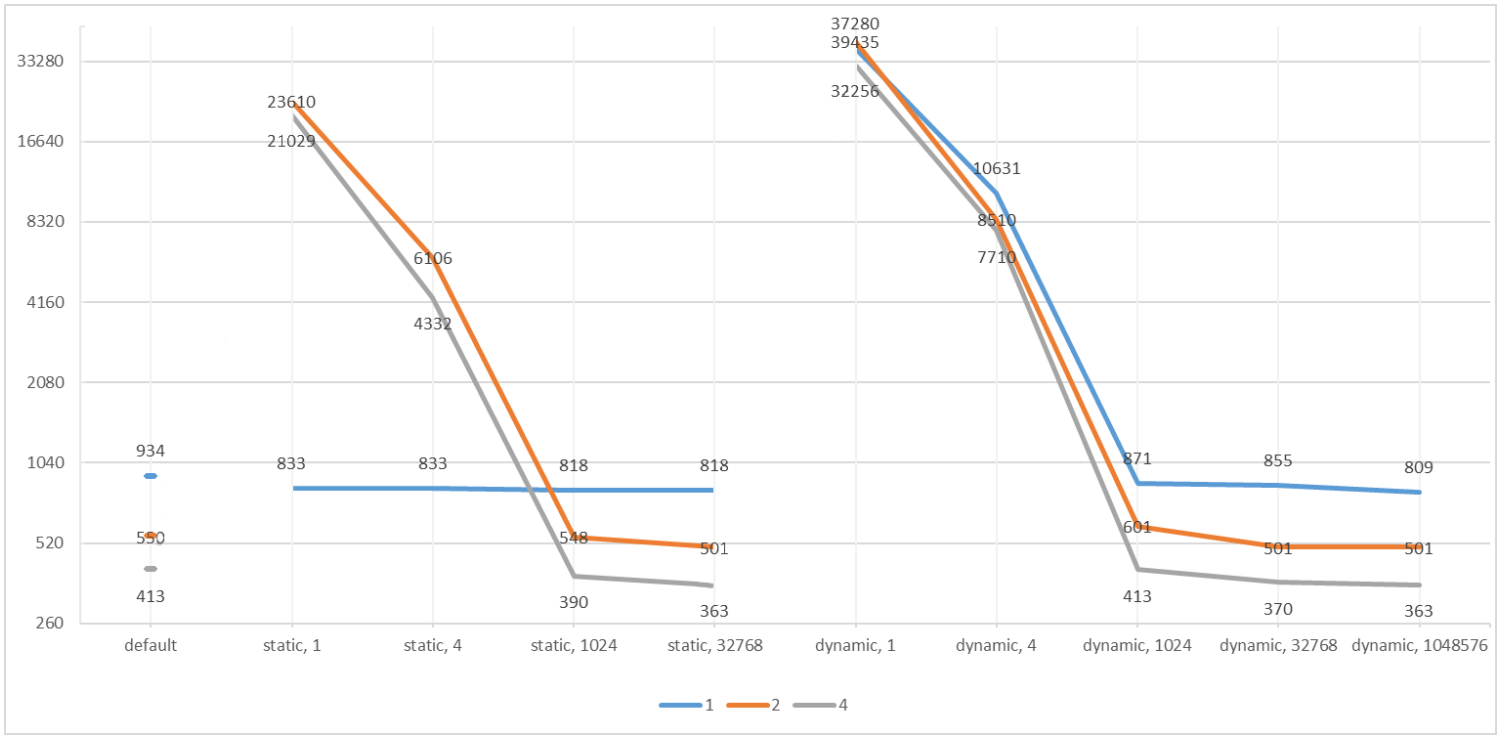

Рисунок 5 – зависимость времени работы от количества потоков при различных параметрах $schedule = \{none, static, dynamic\}$ и различных параметрах $chunk\_size$

На рисунках 3 и 4 показаны зависимости времени работы от количества потоков (по горизонтальной оси), при этом данные предоставлены сразу от нескольких различных параметров $chunk\_size$ (различные линии).

На рисунке 5 показана зависимость времени работы от параметров $schedule$ и $chunk\_size$ (по горизонтальной оси), при этом для различных линий кол-во потоков отличается.

Анализируя график зависимости времени работы от количества потоков при параметре $schedule = static$ (рисунок 3), мы видим, что:

- При одном потоке значение переменной $chunk\_size$ ожидаемо не влияет на скорость работы
- В целом большие значения $chunk\_size$ увеличивают скорость работы программы, ведь мы работаем с циклами очень большого размера.

Анализируя график зависимости времени работы от количества потоков при параметре $schedule = dynamic$ (рисунок 4), мы видим, что:

- Даже при одном потоке значение переменной $chunk\_size$ влияет на скорость работы. Данное свойство требует дальнейшего исследования и обоснования, но интуитивно понятно.
- Аналогично $static$, большие значения $chunk\_size$ увеличивают скорость работы программы, ведь мы работаем с циклами очень большого размера.

Выделяющиеся на фоне других показания при 4 потоках $static, 4$ и 32 потоках d$ynamic, 4$ будем считать ошибкой измерений, погрешностью.

Анализируя график зависимости времени работы от количества потоков при различных параметрах $schedule$ и различных параметрах $chunk\_size$ (рисунок 5), мы видим, что:

- $schedule = default$ ожидаемо даёт хорошие результаты в независимости от кол-ва потоков, выбирая оптимальное значение $\{static, dynamic\}$ и $chunk\_size$

- $schedule = dynamic$ даёт плохие результаты при небольшом значении $chunk\_size$
- $schedule = static$ и $schedule = dynmic$ ожидаемо дают хорошие результаты при больших значениях $chunk\_size$, поскольку мы работаем с большими по размеру циклами.

Подводя итог по параметрам $schedule$ и $chunk\_size$, можно сделать следующие выводы:

- Чем больше $chunk\_size$, тем лучше (в разумных пределах)
- $schedule = static$ (в нашем случае одного простого цикла с известной длительностью выполнения каждой операции) лучше, чем $schedule = dynamic$, однако при больших значениях $chunk\_size$ разница незаметна.
- $schedult = dynamic$ наилучшим образом подходит к данной задаче

Вследствие последнего пункта, для тестирования проекта на максимальной скорости использовались значения $schedule$ и $chunk\_size$ по умолчанию.

Листинг

Компилятор: `g++ (x86_64-posix-seh-rev0, Built by MinGW-W64 project) 8.1.0`

autocontrast.cpp

```
//
//                                   Copyright (c) 2022 Nikita Paniukhin
//                                    Licensed under the MIT license
//
//
//
==========================================================================================
==========================

#pragma GCC optimize("Ofast")
#pragma GCC target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,mmx,tune=native")
#pragma GCC target("avx2")

#include <iostream>
#include <fstream>
#include <sstream>
#include <chrono>
#include <cmath>
#include <omp.h>

using namespace std;



void handle_image(string input_path, string output_path, float coeff, bool debug=false) {
    if (debug) cout << "Handling \"" << input_path << "\"..." << endl;
    chrono::time_point<chrono::high_resolution_clock> start_time, end_time;

    #ifdef _OPENMP
        const int THREADS_COUNT = omp_get_max_threads();
    #else
        const int THREADS_COUNT = 1;
    #endif

    // ================================================ INITIALIZATION
=================================================

    FILE * input = fopen(input_path.c_str(), "rb");

    if (!input) {
        cout << "Error reading input file!" << endl;
        return;
    }

    char first_indentifier, second_indentifier;
    int width, height, color_space;

    if (fscanf(input, "%c%c %d %d %d ", &first_indentifier, &second_indentifier, &width,
&height, &color_space) != 5) {
        cout << "PNM file not recognized" << endl;
        return;
    }

    if (first_indentifier != 'P' || (second_indentifier != '5' && second_indentifier !=
'6')) {
        cout << "PNM file not recognized: \"P5\" or \"P6\" not found" << endl;
        return;
    }
```

```
    bool colored = (second_indentifier == '6');
    int size = width * height, colorwise_size = (colored ? 3 * size : size);

    if (debug) cout << "width: " << width << "\nheight: " << height << "\nsize: " << size
<< endl;


    if (debug) cout << "Allocating memory..." << endl;

    uint8_t *image = (uint8_t *) malloc(sizeof(uint8_t) * colorwise_size);
    if (!image) {
        cout << "Can not allocate memory for this file" << endl;
        return;
    }


    // ================================================== INPUT
==================================================

    if (debug) {
        cout << "Reading file..." << endl;
        start_time = chrono::high_resolution_clock::now();
    }

    fread(image, 1, colorwise_size, input);

    if (debug) {
        end_time = chrono::high_resolution_clock::now();
        cout << "Read in " << chrono::duration_cast<chrono::milliseconds>(end_time -
start_time).count() << "ms" << endl;
    }

    fclose(input);



    // ================================================ PROCESSING
================================================

    if (debug) cout << '\n' << "Processing..." << endl;
    start_time = chrono::high_resolution_clock::now();

    // ------------------------ Frequencies ------------------------
    int thread_block_size = colorwise_size / THREADS_COUNT;
    size_t freq[256] = {0};

    if (debug) start_time = chrono::high_resolution_clock::now();

    #pragma omp parallel
    {
        #ifdef _OPENMP
            int cur_thread_num = omp_get_thread_num();
        #else
            int cur_thread_num = 0;
        #endif

        int start = thread_block_size * cur_thread_num,
            end = thread_block_size * (cur_thread_num + 1);

        size_t tmp_freq[256] = {0};

        // #pragma omp parallel for
        for (int pixel_index = start; pixel_index < end; ++pixel_index) {
```

```
            ++tmp_freq[image[pixel_index]];
        }

        #pragma omp critical
        {
            for (int i = 0; i < 256; ++i) {
                freq[i] += tmp_freq[i];
            }
        }
    }

    if (debug) {
        end_time = chrono::high_resolution_clock::now();
        cout << "Frequences1 in " << chrono::duration_cast<chrono::milliseconds>(end_time -
start_time).count() << "ms" << endl;
        start_time = chrono::high_resolution_clock::now();
    }

    for (int pixel_index = thread_block_size * THREADS_COUNT; pixel_index < colorwise_size;
++pixel_index) {
        ++freq[image[pixel_index]];
    }

    if (debug) {
        end_time = chrono::high_resolution_clock::now();
        cout << "Frequences2 in " << chrono::duration_cast<chrono::milliseconds>(end_time -
start_time).count() << "ms" << endl;
    }

    // --------------------------- Borders ---------------------------
    int source_min, source_max;
    float needed_borders = coeff * size;

    if (debug) start_time = chrono::high_resolution_clock::now();

    // #pragma omp parallel sections
    {
        // #pragma omp section
        {
            size_t pref_summ;
            for (pref_summ = 0, source_min = 0; source_min < 255; ++source_min) {
                pref_summ += freq[source_min];
                if ((float) pref_summ > needed_borders) {
                    pref_summ -= freq[source_min];
                    break;
                }
            }
        }

        // #pragma omp section
        {
            size_t pref_summ;
            for (pref_summ = 0, source_max = 255; source_max > 0; --source_max) {
                pref_summ += freq[source_max];
                if ((float) pref_summ > needed_borders) {
                    pref_summ -= freq[source_max];
                    break;
                }
            }
        }
    }


    if (debug) {
```

```
        end_time = chrono::high_resolution_clock::now();
        cout << "Borders in " << chrono::duration_cast<chrono::milliseconds>(end_time -
start_time).count() << "ms" << endl;
        cout << "min, max = " << (int) source_min << ' ' << (int) source_max << endl;
    }

    // ---------------------------- Processing ----------------------------
    float tmp = (float) 255.0 / (source_max - source_min);

    uint8_t mapping[256];
    if (source_min == source_max) {
        for (int i = 0; i < 256; ++i) mapping[i] = i;
    } else {
        for (int i = 0; i < 256; ++i) {
            mapping[i] = (uint8_t) min(255, (int) round(tmp * max(0, i - source_min)));
        }
    }

    if (debug) start_time = chrono::high_resolution_clock::now();

    #pragma omp parallel for
    for (int pixel_index = 0; pixel_index < colorwise_size; ++pixel_index) {
        image[pixel_index] = mapping[image[pixel_index]];
    }

    end_time = chrono::high_resolution_clock::now();
    float elapsed = ((float) chrono::duration_cast<chrono::microseconds>(end_time -
start_time).count()) / 1000;
    printf("Time (%i thread(s)): %g ms\n", THREADS_COUNT, elapsed);


    // ================================================== OUTPUT
==================================================

    if (debug) {
        cout << '\n' << "Writing output..." << endl;
        start_time = chrono::high_resolution_clock::now();
    }

    FILE * output = fopen(output_path.c_str(), "wb");
    fprintf(output, "P%d\n%d %d\n%d\n", (colored ? 6 : 5), width, height, color_space);
    fwrite(image, 1, colorwise_size, output);
    fclose(output);

    if (debug) {
        end_time = chrono::high_resolution_clock::now();
        cout << "Wrote in " << chrono::duration_cast<chrono::milliseconds>(end_time -
start_time).count() << "ms" << endl;
    }


    // ================================================== THE END
==================================================

    free(image);
    if (debug) cout << '\n' << '\n' << endl;
}

int main(int argc, char* argv[]) {
    #ifdef _OPENMP
        omp_set_nested(1);
    #else
        cout << "Warning: OpenMP is turned off!" << endl;
    #endif
```

```
    // omp_set_num_threads(1);
    // handle_image("images/rgb.pnm", "result/rgb.pnm", 0, true);

    // omp_set_num_threads(72);
    // handle_image("images/picTest9.pnm", "result/picTest9.pnm", 0, true);

    // omp_set_num_threads(1);
    // handle_image("images/rgb.pnm", "result/rgb.pnm", 0, false);
    // for (int thread_cnt = 0; thread_cnt < 8; ++thread_cnt) {
    //     omp_set_num_threads(1 << thread_cnt);
    //     handle_image("images/rgb.pnm", "result/rgb.pnm", 0, false);
    // }
    // return 1;


    if (argc > 1) {
        if (argc < 5) {
            cout << "Too few arguments" << endl;
            return 1;
        }

        istringstream ss1(argv[1]);
        int threads_count;
        if (!(ss1 >> threads_count)) {
            cout << "Invalid number: " << argv[1] << endl;
            return 1;
        } else if (!ss1.eof()) {
            cout << "Trailing characters after number: " << argv[1] << endl;
            return 1;
        }

        istringstream ss2(argv[4]);
        float coeff;
        if (!(ss2 >> coeff)) {
            cout << "Invalid number: " << argv[4] << endl;
            return 1;
        } else if (!ss2.eof()) {
            cout << "Trailing characters after number: " << argv[4] << endl;
            return 1;
        }

        #ifdef _OPENMP
            omp_set_num_threads(threads_count);
        #endif
        handle_image(argv[2], argv[3], coeff);

    } else {
        cout << "No arguments specified, running with debug configuration..." << endl;

        handle_image("images/low_contrast.small.pnm", "result/low_contrast.small.pnm",
0.01, false);
        handle_image("images/low_contrast.large.pnm", "result/low_contrast.large.pnm",
0.01, false);
        handle_image("images/rgb.pnm", "result/rgb.pnm", 0, false);

        for (int i = 0; i <= 12; ++i) {
            if (i == 8) continue;
            handle_image(
                "images/picTest" + to_string(i) + ".pnm",
                "result/picTest" + to_string(i) + ".pnm",
                0
            );
        }
```

```
    }
}
```

run.cmd

```
cls && "C:\Program Files\mingw-w64\x86_64-8.1.0-posix-seh-rt_v6-rev0\mingw64\bin\g++.exe" -std=c++17 -fopenmp autocontrast.cpp -o "hw5.exe" && "hw5.exe"
```