САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Архитектура ЭВМ

Отчет

по домашней работе № 5

«OpenMP»

Выполнил: Захаров Кирилл Витальевич

Номер ИСУ: 334887

студ. гр. М3139

Санкт-Петербург

2021

Цель работы: знакомство со стандартом OpenMP.

Инструментарий и требования к работе: работа была написана на C++ с использованием стандарта OpenMP 2.0.

Теоретическая часть

OpenMP — открытый стандарт для распараллеливания программ на C/C++ и Фортран. В программе ведущий поток может создать несколько ведомых потоков делается это с помощью директивы #pragma omp parallel — таким образом компилятор понимает, что следующий фрагмент кода необходимо распараллелить с помощью OpenMP.

Затем идут директивы и атрибуты того, что мы распараллеливаем. В своей программе я использую две директивы: for и sections. Таким образом, чтобы распараллелить цикл for я использую: #pragma omp parallel for <attributes>. Про атрибуты я скажу позже. Это директива говорит распараллелить цикл for таким образом, чтобы каждая итерация произошла ровно один раз. И чтобы потоки их распределили их между собой (а не только один поток с ними работал), как именно должно произойти распределение указано в атрибутах. #pragma omp parallel sections {<code>} позволяет распределить код на секции, чтобы они выполнялись не одним потоком, а несколькими. Для того, чтобы разделить код на секции используется #pragma omp section внутри кода.

Теперь перейдем к атрибутам. Атрибут shared (variable-list) позволяет указать, какие переменные будут общими у потоков. Однако все переменные по умолчанию являются общими, поэтому данный атрибут можно не писать, кроме тех случаев, когда хочется в явном виде показать, что переменные общие. Это используется, чтобы не было гонок данных, когда потоки имеют копию одной и той же переменной, но делают с ней какую-то операцию и в итоге результат будет иным, если бы программа

выполнялась без распараллеливания. Понятно, что в таком случае ядра будут передавать данные в более медленный кэш (или вообще в оперативную память) для синхронизации общей переменной, и программа из-за этого может начать работать медленней, чем в одном потоке. Поэтому можно использовать атрибут private(variable-list) в нем мы указываем, какие переменные будут в каждом потоке приватными (т. е. у каждого потока будет своя переменная и мы будем об этом знать). Это полезно, т. к. она будет находиться в более быстром кэше и производительность каждого потока не сильно упадет по сравнению с одним потоком без распараллеливания. Аналогичного эффекта можно добиться, если инициализировать переменную внутри #pragma omp parallel блока.

Атрибут num_threads(numThreads) задает количество потоков во время выполнения данной директивы. Параметр schedule(kind, chunk_size) определяет как итерации цикла for должны распределяться между собой. Для сравнения я использовал два вида: 1) static — итерации цикла for будут делиться на блоки размера указанного в chunk_size (если ничего не указано, то итерации разделятся на блоки примерно равного размера) в циклическом порядке, т. е. сначала 1-му потоку, потом 2-му, 3-му и т. д., и опять сначала. 2) dynamic — итерации цикла for будут разделены на блоки размера chunk_size (значение по умолчанию 1), затем блок будет присвоен одному из потоков, свободных в данный момент. Таким образом, если время работы итераций будет непредсказуемым, то dynamic предпочтительней, т. к. потоку, которому присвоился долгий фрагмент, будет присвоено меньше итераций, однако потоки будут немного простаивать во время присвоений фрагментов, и поэтому если работа у всех потоков одинаковая, то лучше использовать static. Атрибут critical нужен, чтобы указать, что эта секция должна исполниться каждым потоком ровно один раз.

Практическая часть

Стандарты .ppm и .pgm довольно просты. В начале файла идут: заголовок (P5 – для .pgm, P6 – для .ppm). Затем идет ширина и высота картинки (числа, закодированные в ASCII) через пробел и максимальное значение цвета (в нашем случае 255). Затем идут данные без шифрования и сжатия. Для чёрно–белых картинок (формат .pgm) 1 пиксель – 1 число, для цветных(формат .ppm) 1 пиксель – 3 числа (в цветом канале RGB). Поэтому нам надо считать заголовок файла, а затем по считать массив цветов (я использовал unsigned char – 8-битные числа). Для .pgm – размер будет height * width, для .ppm – размер будет 3 * height * width. Нам на вход дан коэффициент — доля самых светлых и самых тёмных, которые надо проигнорировать, поэтому наша цель – посчитать, количество пикселей каждого оттенка от 0 до 255 и найти минимальный оттенок, что число пикселей темнее этого оттенка (low) >= коэффициент * число пикселей. Аналогично и со светлыми пикселями (high). Замечу, что алгоритм выглядит одинаково для RGB и для чёрно–белых картинок, просто в RGB мы для каждого канал отдельно находим low и high, и среди low мы берем самый наименьший; среди high, самый наибольший. У меня есть массив rgb[3][256](для цветных) или grey[256] (для чёрно-белых). Для того, чтобы эффективно распараллелить подсчёт каждого оттенка, давайте сделаем следующее: создадим локальный массив localrgb[3][256] (или localgrey[256]), в нем посчитаем количество каждого цвета, т. е. localrgb[i % 3][picture[i]]++ (или localgrey[picture[i]]++), массив picture – массив цветов, и затем сольем это в critical секции с массивом rgb или grey. Таким образом, локальные массивы будут созданы в более быстром кэше (т. к. в случае, если бы было просто rgb[i % 3][picture[i]]++, мы бы во время всех итераций ходили в медленный кэш и программа бы работала медленней, чем в одном потоке), и мы будем идти в медленный кэш только в critical секции.

Затем я для каждого канала RGB, нашел самый темный цвет, и самый светлый цвет, которые должны будут перейти в 0 и 255 соответственно, я это распараллелил с использованием секций (sections), т. к. в этом случае каналы не связаны.

Растяжение диапазона я делал с помощью линейного преобразования для каждого числа отдельно. Алгоритм: есть цвет, если он светлее high, то должен стать 255, если темнее low, то должен стать 0, иначе делаем с ним следующее:

$$picture[i] = \frac{(picture[i] - low) * 255}{high - low}$$

Понятно, что low перейдет в 0, high – в 255, а значит, т. к. функция линейна, все пиксели будут линейно преобразовываться. Теперь про округление. Можно приводить к float и округлять, но из-за этого время работы увеличивается значительно, можно заметить, что все числа целые, а поэтому для округления к ближайшему целому можно к числителю прибавить:

$$\frac{high - low}{2}$$

И это будет работать при обычном целочисленном делении. Всё это делается в #pragma omp parallel for с общим массивом picture, т. к. здесь очевидно нет смысла создавать локальные массивы.

Сравнение времени работы программы

Для сравнения я запускал программу 10 раз на картинке размером около 200 мегабайт и считал среднее время работы программы, т. к. есть операционная система, которая по-разному может распределять ресурсы, непостоянные частоты процессора, промахи в кэше и другие факторы,

которые незначительно влияют на время работы. Тесты будут ухудшаться после 12 потоков, т. к. у меня 12–поточный процессор и делить программу на большее число потоков уже невыгодно.

Сначала сравним, с одинаковым параметром в schedule, но разным числом потоков. На графике 1 представлено сравнение static с размером 1 и 64. У всех графиков x координата — количество потоков, y координата — время в мс.

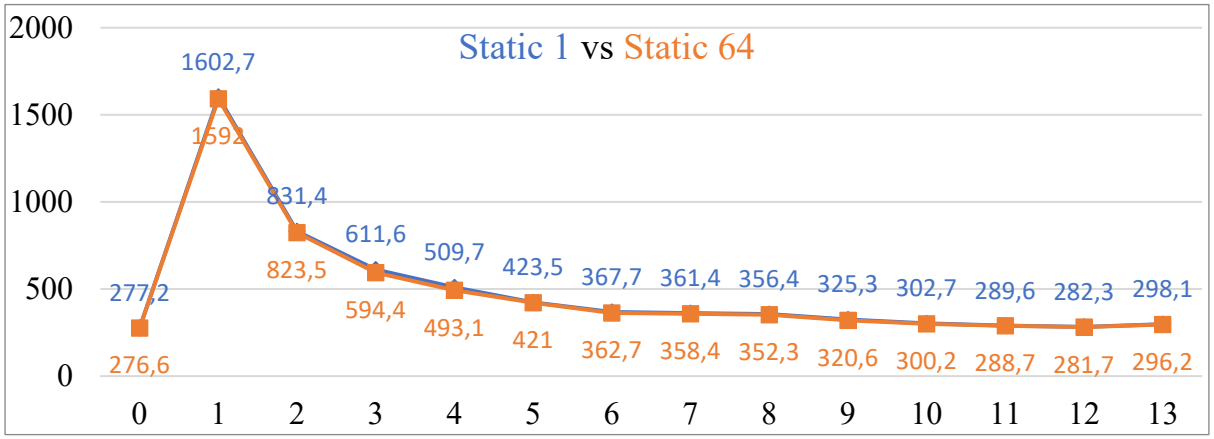

График 1 — сравнение static 1 и static 64

На графике 2 представлено сравнение static с размером по умолчанию и 256.

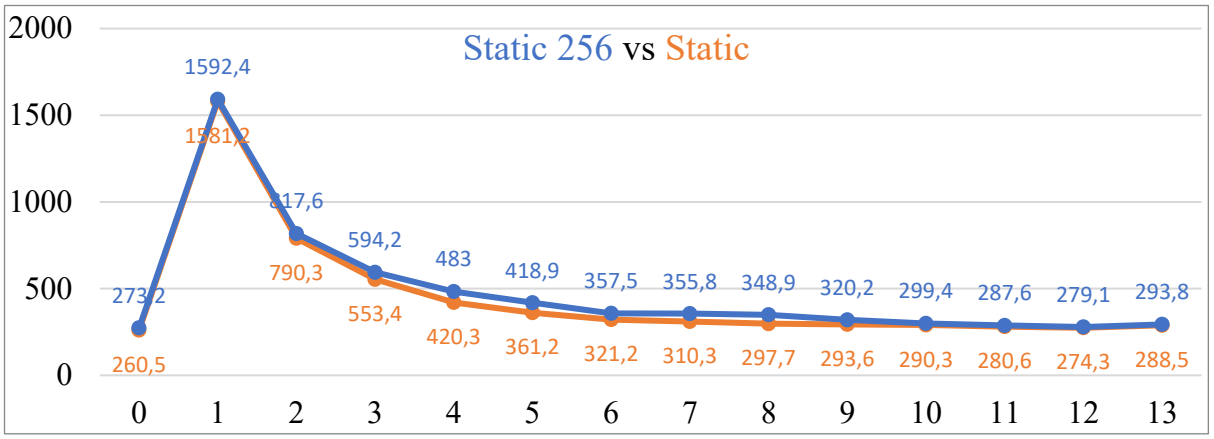

График 2 — сравнение static 256 и static

Заметим, что наилучшую производительность показывает static со значением по умолчанию. Вполне ожидаемый результат, т. к. при значении

static по умолчанию каждый поток только один раз создает локально массив, и передает свои локальные данные в медленный кэш.

Теперь сравним dynamic секции. Для начала проверим, действительно ли значение по умолчанию в dynamic секции равно 1. Доказывает нам это таблица 3.

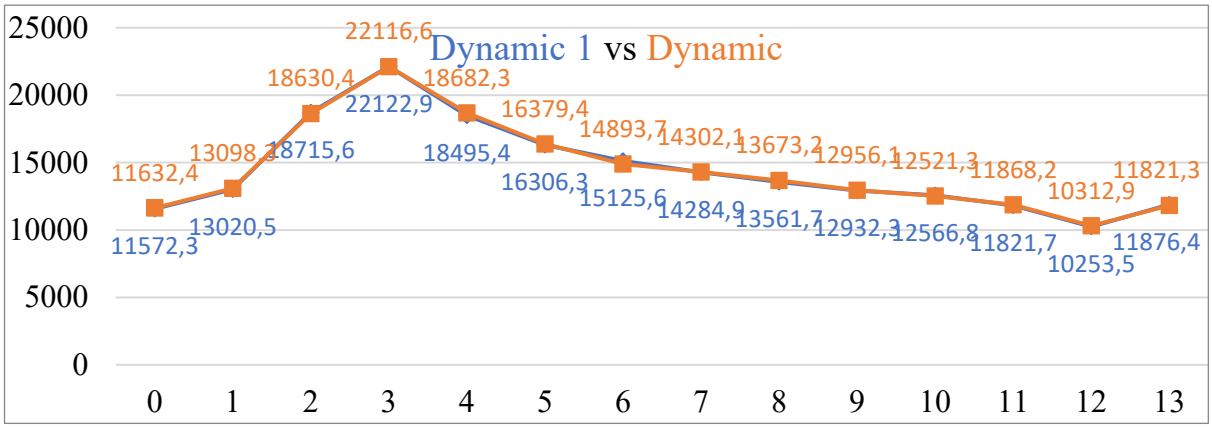

График 3 — сравнение dynamic 1 и dynamic

Данный результат был ожидаем, т. к. при dynamic происходит простаивание потоков, и при этом время каждого потока примерно одинаковое. Теперь сравним Dynamic друг с другом. Очевидно, что размер 1 проигрывает, поэтому сравним 64 и 256. Результаты – график 4.

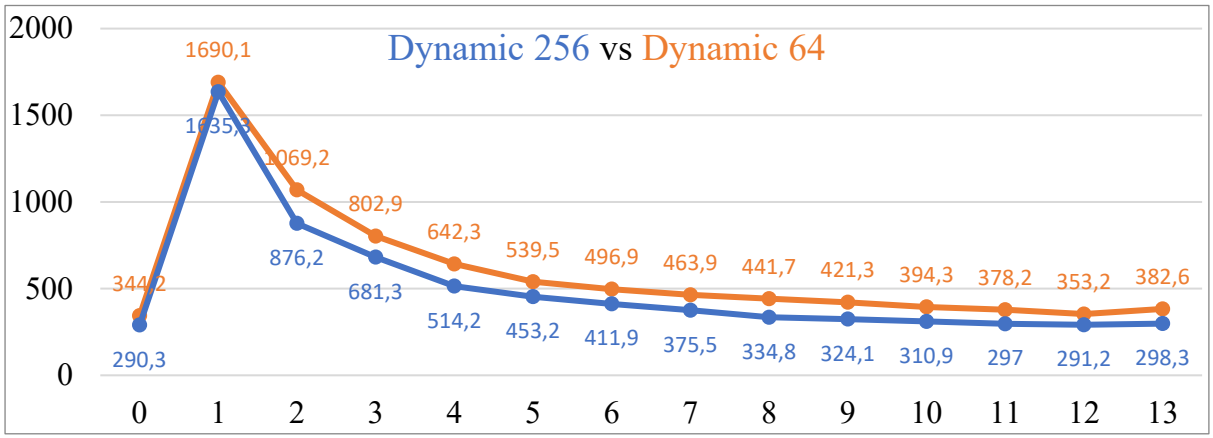

График 4 — сравнение Dynamic 64 и Dynamic 256

Тоже вполне ожидаемый результат, т. к. больше размер — меньше время простаивания каждого потока. Теперь сравним Dynamic 256 и Static. Результат график 5.

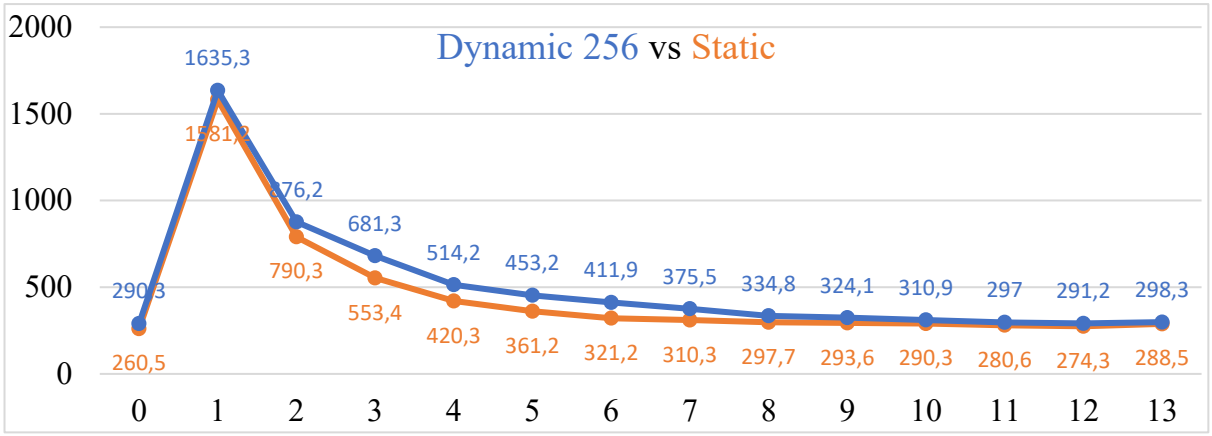

График 5 — сравнение dynamic 256 и static

Понятно, почему static выигрывает. Теперь осталось сравнить код без использования OpenMP с использованием static и кодом, использущим OpenMP, но в одном потоке. Результаты – график 6.

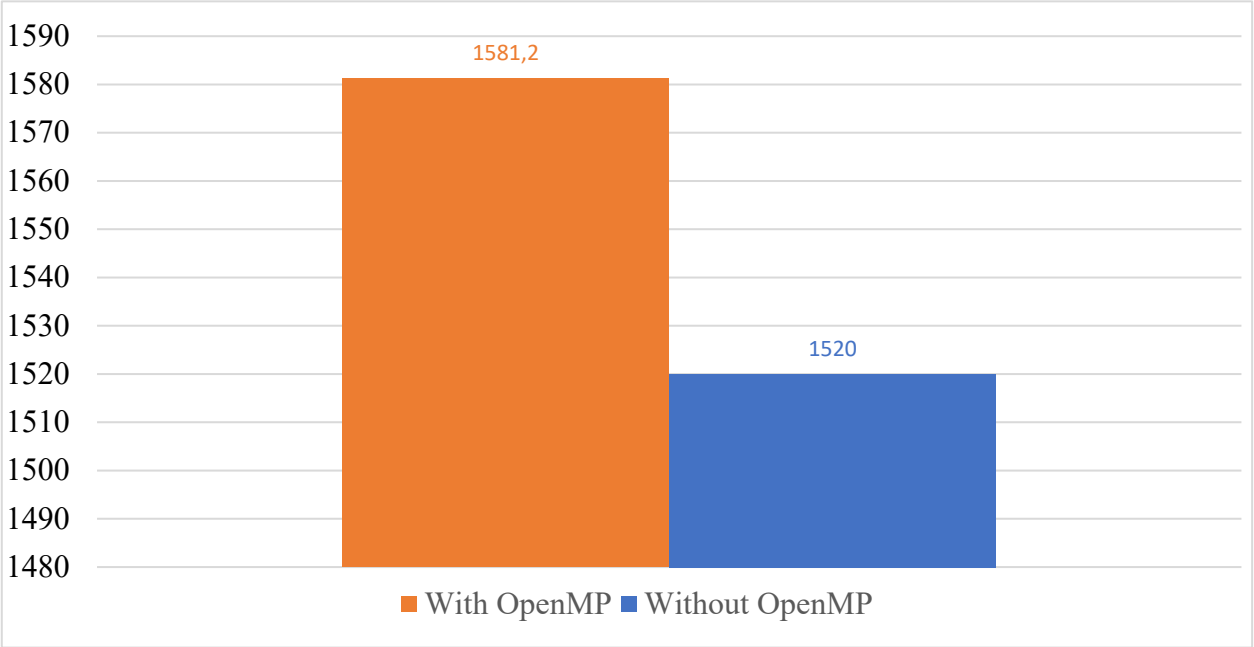

График 6 — сравнение программы с OpenMP и без OpenMP

Листинг

Я использовал C++ и компилировал на mingw-w64 9.0 с указанием флага компиляции -fopenmp в файле CMake.

main.cpp

```cpp
#include <iostream>
#include <cstdlib>
#include "omp.h"
#include <cmath>
#include <chrono>

#pragma GCC optimize ("O3")

using namespace std;

#define RUN_TYPE static

FILE *inputPicture;
unsigned int count;
float coefficient;
unsigned int val;

unsigned int getInt() {
    unsigned int ret = 0;
    char ch;
    fscanf(inputPicture, "%c", &ch);
    while (!isspace(ch)) {
        ret *= 10;
        ret += (ch - '0');
        fscanf(inputPicture, "%c", &ch);
    }
    return ret;
}

pair<unsigned char, unsigned char> getLowHigh(int a[]) {
    int high = 255;
    unsigned int sum = 0;
    for (; high >= 0; high--) {
        if (sum + a[high] > val) {
            break;
        }
        sum += a[high];
    }
    int low = 0;
    sum = 0;
    for (; low <= 255; low++) {
        if (sum + a[low] > val) {
            break;
        }
        sum += a[low];
    }
    return {low, high};
}

int main(int argc, char *argv[]) {
    int numThreads = stoi(argv[1]);
    inputPicture = fopen(argv[2], "rb");
    unsigned char type;
    char ch;
    fscanf(inputPicture, "%c%c", &type, &type);
    fscanf(inputPicture, "%c", &ch);
    unsigned int width = getInt();
    unsigned int prevWidth = width;
    unsigned int height = getInt();
```

```
    unsigned int maxValue = getInt();
    coefficient = stof(argv[4]);
    count = width * height;
    val = round(count * coefficient);
    if (type == '6') {
        width *= 3;
        count *= 3;
    }
    unsigned char *picture = (unsigned char *) malloc(width * height *
sizeof(unsigned char *));
    fread(picture, 1, count, inputPicture);
    fclose(inputPicture);
    auto begin = std::chrono::high_resolution_clock::now();
    if (type == '5') {
        int grey[256] = {0};
        #pragma omp parallel shared(grey, count) num_threads(numThreads)
        {
            int localgrey[256] = {0};
            #pragma omp for schedule(RUN_TYPE)
            for (int i = 0; i < count; i++) {
                localgrey[picture[i]]++;
            }
            #pragma omp critical
            for (int i = 0; i < 256; i++) {
                grey[i] += localgrey[i];
            }
        }
        pair<unsigned char, unsigned char> pair1 = getLowHigh(grey);
        unsigned char low = pair1.first;
        unsigned char high = pair1.second;
        unsigned char dop = (high - low) / 2;
        #pragma omp parallel for num_threads(numThreads) schedule(RUN_TYPE)
        for (int i = 0; i < count; i++) {
            if (picture[i] >= high) {
                picture[i] = 255;
                continue;
            }
            if (picture[i] <= low) {
                picture[i] = 0;
                continue;
            }
            picture[i] = (unsigned char) ((picture[i] - low) * 255 + dop) / (high
- low);
        }
    } else {
        int rgb[3][256];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 256; j++) {
                rgb[i][j] = 0;
            }
        }
        #pragma omp parallel shared(rgb, count) num_threads(numThreads)
        {
            int localrgb[3][256];
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 256; j++) {
                    localrgb[i][j] = 0;
                }
            }
```

```
            #pragma omp for schedule(RUN_TYPE)
            for (int i = 0; i < count; i++) {
                localrgb[i % 3][picture[i]]++;
            }
            #pragma omp critical
            for (int i = 0; i < 3; i++) {
                for (int j = 0; j < 256; j++) {
                    rgb[i][j] += localrgb[i][j];
                }
            }
        }
        pair<unsigned char, unsigned char> lowHighRed;
        pair<unsigned char, unsigned char> lowHighGreen;
        pair<unsigned char, unsigned char> lowHighBlue;
        #pragma omp parallel sections num_threads(numThreads)
        {
            #pragma omp section
            lowHighRed = getLowHigh(rgb[0]);
            #pragma omp section
            lowHighGreen = getLowHigh(rgb[1]);
            #pragma omp section
            lowHighBlue = getLowHigh(rgb[2]);
        }
        unsigned char lowRGB = min(lowHighRed.first, min(lowHighGreen.first,
lowHighBlue.first));
        unsigned char highRGB = max(lowHighRed.second, max(lowHighGreen.second,
lowHighBlue.second));
        int kRGB = highRGB - lowRGB;
        unsigned char dop = kRGB / 2;
        #pragma omp parallel for num_threads(numThreads) schedule(RUN_TYPE)
        for (int i = 0; i < count; i++) {
            if (picture[i] >= highRGB) {
                picture[i] = 255;
                continue;
            }
            if (picture[i] <= lowRGB) {
                picture[i] = 0;
                continue;
            }
            picture[i] = (unsigned char) ((((picture[i] - lowRGB) * 255 + dop) /
kRGB));
        }
    }
    auto end = std::chrono::high_resolution_clock::now();
    double duration = std::chrono::duration_cast<std::chrono::milliseconds>(end -
begin).count();
    FILE *outputPicture = fopen(argv[3], "wb");
    fprintf(outputPicture, "P%c\n%d %d\n%d\n", type, prevWidth, height, maxValue);
    fwrite(picture, 1, count, outputPicture);
    fclose(outputPicture);
    printf("Time (%i thread(s)): %g ms\n", numThreads, duration);
}
```