САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: OpenCL

Отчет

по лабораторной работе № 1

**«OpenMP. Вычисление гистограммы изображения»**

Выполнил: Акимов Николай Сергеевич

Номер ИСУ: 264328

студ. гр. M33381

Санкт-Петербург

2021

**Цель работы:** изучить алгоритм и реализовать программу, выполняющую подсчет гистограммы для серых изображений в формате PNM с использованием OpenMP.

**Инструментарий и требования к работе:**Программа должна быть написана на C/C++ без использования внешних библиотек за исключением библиотеки OpenMP.

**Практическая часть**

Изображение в формате PNM можно представить как массив пикселей, поэтому задачу подсчёта гистограммы можно разбить на несколько потоков путём разбиения массива на несколько равных частей и обработкой каждым потоком отдельной части.

При подсчёте гистограммы каждый поток локально заводит массив на 256 элементов, итерируется по соответствующему куску массива, после подсчёта гистограммы на отдельном участке массива надо корректно обновить общий массив. Для этого есть несколько вариантов: либо у нас есть массив атомарных переменных, либо для того, чтобы обновить общий массив на 256 элементов надо войти в критическую секцию. Был выбран второй способ, поскольку накладных расходов на 256 атомарных прибавлений больше, чем просто войти в критическую секцию и довольно быстро сделать обычных 256 прибавлений.

Конфигурация ПК:

* Процессор: Intel® Core™ i5-6200U CPU @ 2.30GHz
* ОЗУ: 6 ГБ

Ниже представлены результаты работы алгоритма на изображении, состоящем из 500 000 000 пикселей. На рисунке 1 представлен график времени работы алгоритма от количества потоков при статическом планировщике. На графике видно, что однопоточная версия работает быстрее, чем многопоточная с одним потоком, потому что однопоточная версия не тратит время работы на создание потоков, а также на конечную синхронизацию. Также на графике видно, что максимальный выигрыш достигается при четырёх потоках, однако выигрыш составляет чуть больше, чем в два раза. Это связано с тем, что на тестируемом ПК имеется всего два реальных ядра, поэтому максимум два потока могут иметь независимые кэши, при количестве потоков больше двух кэш-подсистеме приходится постоянно вытеснять данные двух потоков, чтобы они могли работать вместе.

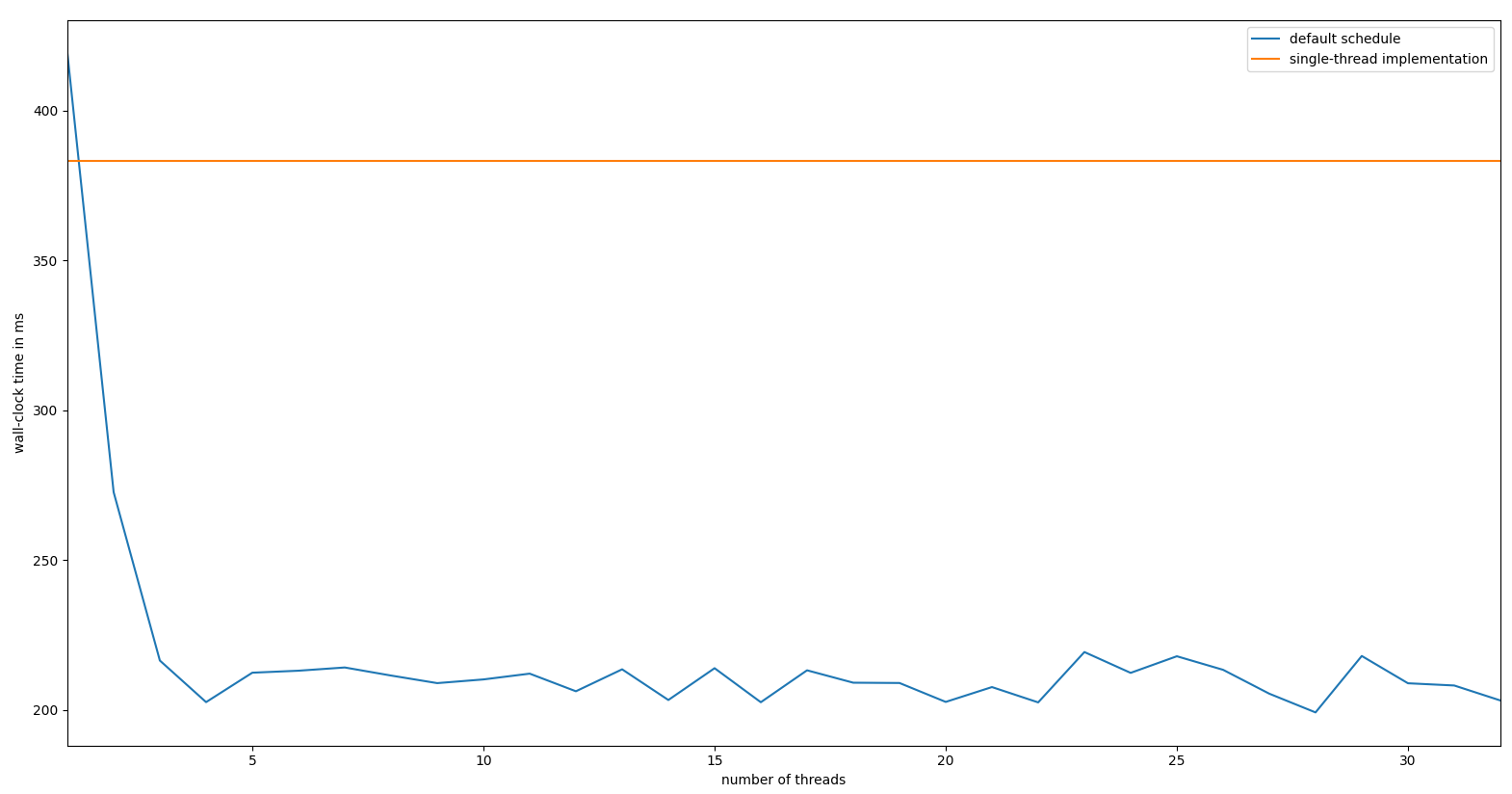


Рисунок 1 – время работы при статическом планировании

Если вместо статического планировщика использовать простой динамический, то время работы заметно увеличивается. Это связано с тем, что каждая итерация цикла выполняет простые операции, поэтому накладные расходы на выбор, какой из потоков будет обрабатывать следующий пиксель, превышает выигрыш от многопоточности. Также потоки работают в разных ядрах, поэтому если они обращаются к одной кэш-линии, то процессору также приходится копировать одни и те же данные разным ядрам, что также неэффективно. На рисунке 2 представлен график времени работы от количества потоков. Также видно, что выгоднее всего запускать всего один поток, но по сравнению с однопоточной реализацией многопоточная работает дольше во много раз больше.

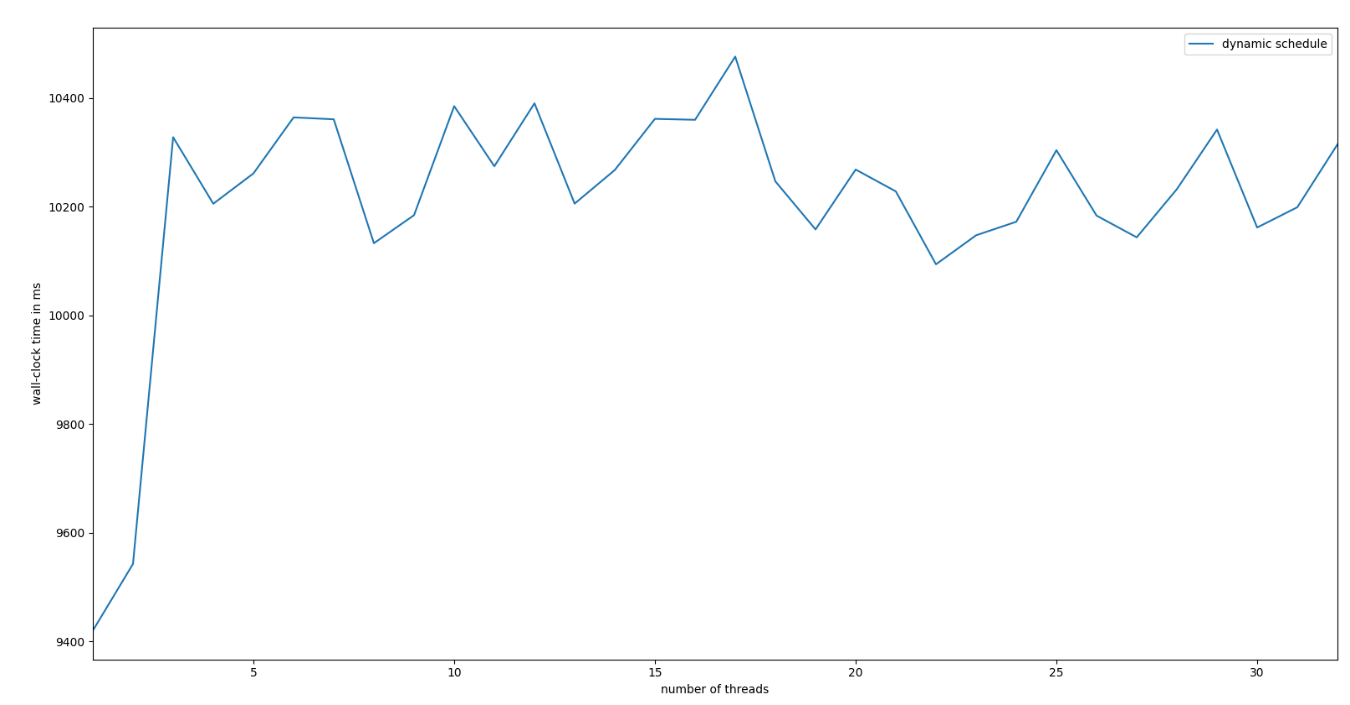


Рисунок 2 – время работы при динамическом планировании

Однако если позволит планировщику давать потоку не по одному пикселю, а по 256, то время работы заметно уменьшается, однако не достигает лучшего результат при статическом подходе. Это связано с тем, что все операции выполняются примерно за одно и то же время, поэтому получить выигрыш с динамического планировщика очень сложно. На рисунке 3 видно, что время работы значительно улучшилось по сравнению с предыдущим графиком.

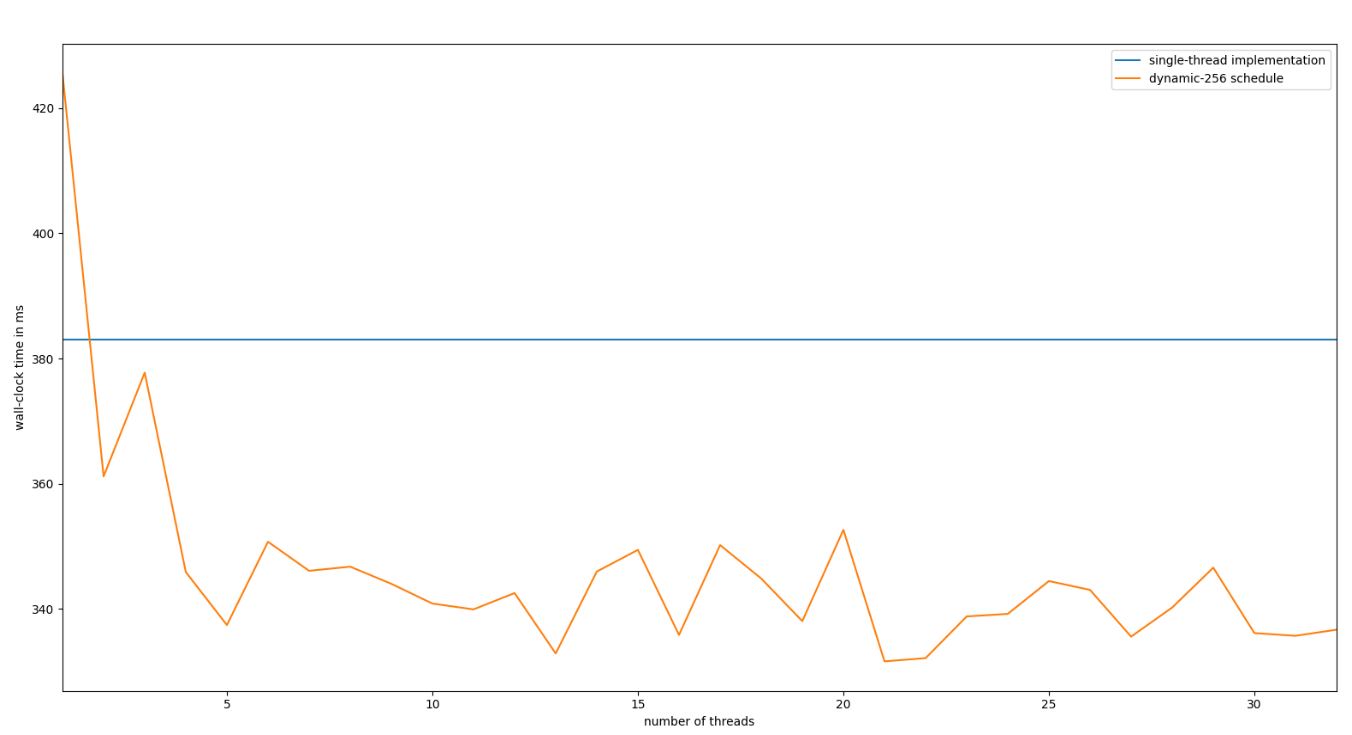


Рисунок 3 – время работы при динамическом планировании по 256 пикселей

**Листинг**

**main.cpp**

#include <iostream>

#include <fstream>

#include <cstdint>

#include <string>

#include <cstdio>

#include <chrono>

#include <omp.h>

#include "pgm\_image.h"

size\_t get\_threads(int32\_t threads) {

if (threads == -1) return 1;

if (threads == 0) return omp\_get\_max\_threads();

return threads;

}

int main(int argc, char\* argv[]) {

if (argc != 4) {

std::cerr << "Input format: <input file> <output file> <num of threads>" << std::endl;

return 1;

}

int32\_t threads;

{

try {

size\_t n;

threads = std::stoi(argv[3], &n);

if (argv[3][n] != '\0') throw std::runtime\_error("Expected number");

} catch (...) {

std::cerr << "Expected the number as a third argument" << std::endl;

return 1;

}

if (threads < -1) {

std::cerr << "Expected the number not less than -1" << std::endl;

return 1;

}

}

uint32\_t buf[256];

try {

pgm\_image image(argv[1]);

auto t\_start = std::chrono::high\_resolution\_clock::now();

if (threads == -1) {

image.get\_histogram(buf);

} else {

image.get\_histogram(buf, threads);

}

auto t\_end = std::chrono::high\_resolution\_clock::now();

std::cout << std::endl << "Time (" << get\_threads(threads) << " thread(s)): " << std::chrono::duration<double, std::milli>(t\_end - t\_start).count() << " ms" << std::endl;

} catch (std::exception& e) {

std::cerr << e.what() << std::endl;

return 1;

}

std::ofstream output(argv[2], std::ios\_base::binary);

if (output.fail()) {

std::cerr << "Could not open file to write" << std::endl;

return 1;

}

output.write(reinterpret\_cast<char\*>(&buf), 1024);

if (output.fail()) {

std::cerr << "Could not open file to write" << std::endl;

std::remove(argv[2]);

return 1;

}

return 0;

}

**pgm\_image.h**

#ifndef PGM\_IMAGE\_H

#define PGM\_IMAGE\_H

#include <memory>

#include <cstdint>

#include <string>

#include <array>

struct pgm\_image {

pgm\_image(std::string const& filename);

pgm\_image(pgm\_image const&) = delete;

pgm\_image& operator=(pgm\_image const&) = delete;

~pgm\_image() = default;

void get\_histogram(uint32\_t (&buf)[256]);

void get\_histogram(uint32\_t(&buf)[256], size\_t num\_threads);

private:

std::unique\_ptr<uint8\_t[]> data;

uint32\_t w;

uint32\_t h;

uint32\_t depth;

};

#endif

**pgm\_image.cpp**

#include <fstream>

#include <exception>

#include <omp.h>

#include "pgm\_image.h"

static uint32\_t get\_number(char const\* s) {

char const\* cur = s;

if (\*cur == '\0') throw std::runtime\_error("Incorrect format of file");

while (\*cur != '\0') {

if (!std::isdigit(\*cur)) throw std::runtime\_error("Incorrect format of file");

cur++;

}

uint32\_t num = std::stoul(s);

if (num == 0) throw std::runtime\_error("Incorrect format of file");

return num;

}

pgm\_image::pgm\_image(std::string const& filename) {

std::ifstream input(filename, std::ios\_base::binary);

if (input.fail()) throw std::runtime\_error("Could not open input file");

char input\_str[128];

input.get(input\_str, 128, '\n');

if (strcmp(input\_str, "P5") != 0) throw std::runtime\_error("Incorrect format of file, expected P5");

input.ignore();

input.get(input\_str, 128, ' ');

w = get\_number(input\_str);

input.ignore();

input.get(input\_str, 128, '\n');

h = get\_number(input\_str);

input.ignore();

input.get(input\_str, 128, '\n');

depth = get\_number(input\_str);

size\_t length = static\_cast<size\_t>(w) \* h;

try {

data = std::unique\_ptr<uint8\_t[]>(new uint8\_t[length]);

} catch (...) {

throw std::runtime\_error("Could not allocate memory");

}

input.ignore();

input.read(reinterpret\_cast<char\*>(data.get()), length);

if (input.fail()) throw std::runtime\_error("Incorrect format of file");

input.ignore();

if (!input.eof()) throw std::runtime\_error("Incorrect format of file");

}

void pgm\_image::get\_histogram(uint32\_t(&buf) [256]) {

for (size\_t i = 0; i < 256; i++) buf[i] = 0;

for (size\_t i = 0; i < static\_cast<size\_t>(w) \* h; i++) {

buf[data[i]]++;

}

}

void pgm\_image::get\_histogram(uint32\_t(&buf)[256], size\_t num\_threads) {

for (size\_t i = 0; i < 256; i++) buf[i] = 0;

if (num\_threads != 0) omp\_set\_num\_threads(num\_threads);

#pragma omp parallel

{

uint32\_t cur\_buf[256];

for (size\_t j = 0; j < 256; j++) cur\_buf[j] = 0;

#pragma omp for

for (int i = 0; i < static\_cast<int>(w) \* h; i++) {

cur\_buf[data[i]]++;

}

#pragma omp critical

{

for (size\_t j = 0; j < 256; j++) buf[j] += cur\_buf[j];

}

}

}