Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

Студент: Ветошкина София Владимировна

Группа: М8О-203Б-23

Вариант: 13

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_\_

Москва, 2024

**Содержание**

1. Репозиторий
2. Постановка задачи
3. Общие сведения о программе
4. Метод и алгоритм решения задачи
5. Исходный код
6. Демонстрация работы программы
7. Выводы

**Репозиторий**

<https://github.com/sofiavetoshkina/os_labs/tree/main>

**Постановка задачи**

Составить программу, обрабатывающую данные в многопоточном режиме. При обработки использовать стандартные средства создания потоков операционной системы. Ограничение максимального количества потоков, работающих в один момент времени, должно быть задано ключом запуска программы.

**Общие сведения о программе**

Программой реализовано наложение K раз фильтра, использующего матрицу свертки, на матрицу, состоящую из вещественных чисел. Размер окна задается пользователем. При ее запуске в качестве аргумента командной строки необходимо указать максимальное количество используемых потоков.

Программа собирается системой сборки CMake.

Реализованы тесты для проверки корректности программы с помощью Google Test, а также благодаря им оценено ускорение и эффективность алгоритма от входных данных и количества потоков.

**Метод и алгоритм решения задачи**

Считывается исходная матрица данных, матрица ядра (в нашем случае в тестах ядро нормированно, то есть сумма элементов равна 1), количество итераций наложения фильтра, а также максимальное количество потоков. Общая матрица разбивается на блоки строк, которые обрабатываются разными потоками, для создания и управления потоками используется pthread. Передача данных в потоки осуществляется через структуру threadArgs. Алгоритм применения свертки осуществляется следующим образом: на каждый элемент исходной матрицы накладывается ядро и считается сумма умножений элементов этого ядра на соответствующие элементы матрицы; eсли элемент ядра выходит за пределы матрицы, то исходная матрица как бы «обрастает» нулями. Пример наложения фильтра можно посмотреть в реализации lab2\_test.cpp. Обновление результирующей матрицы выполняется после каждой итерации. Потоки создаются в ограниченном количестве, чтобы не перегружать процессор и не вызывать избыточного переключения контекста. После завершения работы потоки синхронизируются с помощью функции pthread\_join. Происходит только чтение элементов матрицы, запись же осуществляется в другую результирующую матрицу. Чтобы сравнить результат многопоточной реализации с однопоточной версией, написаны google tests. В нашем случае проводятся тесты производительности, чтобы убедиться, что многопоточная реализация работает быстрее.

Таким образом, алгоритм решения задачи: разделяем исходную матрицу на блоки строк; каждый поток обрабатывает свой блок, выполняя свертку для каждой строки; собираем результаты всех потоков в результирующую матрицу; повторяем процесс заданное количество раз. Стоит отметить, что эффективнее разделить матрицу на строки, так как создавать поток на каждый элемент матрицы затратнее.

**Исходный код**

lab2/main.cpp:

#include <iostream>

#include "filter.h"

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

if (argc != 2) {

std::cerr << "Введите: " << argv[0] << " <count\_threads>\n";

return 1;

}

int countThreads = std::stoi(argv[1]);

int rows, cols, kernelSize, iterations;

std::cout << "Введите <rows> <cols> <kernelSize> <count\_of\_filter>: " << std::endl;

std::cin >> rows >> cols >> kernelSize >> iterations;

TMatrix matrix(rows, std::vector<double>(cols));

TMatrix kernel(kernelSize, std::vector<double>(kernelSize));

std::cout << "Введите вещественную матрицу: " << std::endl;

for (int i = 0; i < rows; ++i)

for (int j = 0; j < cols; ++j)

std::cin >> matrix[i][j];

std::cout << "Введите матрицу свертки(нормированность соблюдается): " << std::endl;

for (int i = 0; i < kernelSize; ++i)

for (int j = 0; j < kernelSize; ++j)

std::cin >> kernel[i][j];

TMatrix result = ApplyConvolution(matrix, kernel, iterations, countThreads);

std::cout << "Результирующая матрица после наложения фильтра K раз: " << std::endl;

for (const auto& row : result) {

for (double value : row) {

std::cout << value << " ";

}

std::cout << '\n';

}

return 0;

}

lab2/include/filter.h:

#pragma once

#include <vector>

using TMatrix = std::vector<std::vector<double>>;

TMatrix ApplyConvolution(const TMatrix& matrix, const TMatrix& kernel, int iterations, int countThreads);

lab2/src /filter.cpp:

#include <algorithm>

#include <iostream>

#include <pthread.h>

#include <vector>

#include "filter.h"

struct threadArgs {

const TMatrix\* matrix;

const TMatrix\* kernel;

TMatrix\* resultMatrix;

int startRow;

int endRow;

int kernelSize;

};

void\* ApplyConvolutionToRows(void\* args) {

threadArgs\* params = (threadArgs\*)(args);

const TMatrix& matrix = \*params->matrix;

const TMatrix& kernel = \*params->kernel;

TMatrix& resultMatrix = \*params->resultMatrix;

int kernelSize = params->kernelSize;

int halfKernel = kernelSize / 2;

for (int i = params->startRow; i < params->endRow; ++i) {

for (int j = 0; j < (int)matrix[0].size(); ++j) {

double value = 0.0;

for (int ki = 0; ki < kernelSize; ++ki) {

for (int kj = 0; kj < kernelSize; ++kj) {

int ni = i + ki - halfKernel;

int nj = j + kj - halfKernel;

if (ni >= 0 && ni < (int)matrix.size() && nj >= 0 && nj < (int)matrix[0].size()) {

value += matrix[ni][nj] \* kernel[ki][kj];

}

}

}

resultMatrix[i][j] = value;

}

}

return nullptr;

}

TMatrix ApplyConvolution(const TMatrix& matrix, const TMatrix& kernel, int iterations, int countThreads) {

int rows = matrix.size();

int cols = matrix[0].size();

int kernelSize = kernel.size();

TMatrix current = matrix;

TMatrix next(rows, std::vector<double>(cols));

for (int iter = 0; iter < iterations; ++iter) {

std::vector<pthread\_t> threads(countThreads);

std::vector<threadArgs> args(countThreads);

int baseRowsPerThread = rows / countThreads;

int extraRows = rows % countThreads;

int currentRow = 0;

for (int t = 0; t < countThreads; ++t) {

int threadRows = baseRowsPerThread + (t < extraRows ? 1 : 0);

args[t] = {

&current,

&kernel,

&next,

currentRow,

currentRow + threadRows,

kernelSize

};

pthread\_create(&threads[t], nullptr, ApplyConvolutionToRows, &args[t]);

currentRow += threadRows;

}

for (int t = 0; t < countThreads; ++t) {

pthread\_join(threads[t], nullptr);

}

std::swap(current, next);

}

return current;

}

tests/lab2\_test.cpp:

#include <chrono>

#include <gtest/gtest.h>

#include <vector>

#include "filter.h"

TEST(ConvolutionTest, SingleThreadCorrectness) {

TMatrix matrix = {

{10.0, 10.0, 10.0},

{10.0, 10.0, 10.0},

{10.0, 10.0, 10.0}

};

// Принимаем, что ядро нормированно(сумма всех элементов = 1)

TMatrix kernel = {

{0.1, 0.1, 0.1},

{0.1, 0.2, 0.1},

{0.1, 0.1, 0.1}

};

TMatrix expected = {

{5.0, 7.0, 5.0},

{7.0, 10.0, 7.0},

{5.0, 7.0, 5.0}

};

int countThreads = 1;

int iterations = 1;

TMatrix result = ApplyConvolution(matrix, kernel, iterations, countThreads);

for(size\_t i = 0; i < matrix.size(); i++) {

for(size\_t j = 0; j < matrix[0].size(); j++) {

EXPECT\_DOUBLE\_EQ(expected[i][j], result[i][j]);

}

}

}

TEST(ConvolutionTest, SingleThreadCorrectness2) {

TMatrix matrix = {

{10.0, 10.0, 10.0},

{10.0, 10.0, 10.0},

{10.0, 10.0, 10.0}

};

// Принимаем, что ядро нормированно(сумма всех элементов = 1)

TMatrix kernel = {

{0.1, 0.1, 0.1},

{0.1, 0.2, 0.1},

{0.1, 0.1, 0.1}

};

TMatrix expected = {

{3.4, 4.8, 3.4},

{4.8, 6.8, 4.8},

{3.4, 4.8, 3.4}

};

int countThreads = 1;

int iterations = 2;

TMatrix result = ApplyConvolution(matrix, kernel, iterations, countThreads);

for(size\_t i = 0; i < matrix.size(); i++) {

for(size\_t j = 0; j < matrix[0].size(); j++) {

EXPECT\_DOUBLE\_EQ(expected[i][j], result[i][j]);

}

}

}

TEST(ConvolutionTest, MultiThreadIsSameAsSingleThread) {

TMatrix largeMatrix(100, std::vector<double>(100, 1.0));

TMatrix kernel = {

{0.0, -1.0, 0.0},

{-1.0, 5.0, -1.0},

{0.0, -1.0, 0.0}

};

int iterations = 1;

TMatrix result = ApplyConvolution(largeMatrix, kernel, iterations, 1);

TMatrix resultMulti = ApplyConvolution(largeMatrix, kernel, iterations, 4);

for(size\_t i = 0; i < largeMatrix.size(); i++) {

for(size\_t j = 0; j < largeMatrix[0].size(); j++) {

EXPECT\_DOUBLE\_EQ(result[i][j], resultMulti[i][j]);

}

}

}

TEST(ConvolutionTest, MultiThreadIsFasterThanSingleThread) {

TMatrix largeMatrix(1000, std::vector<double>(1000, 0.1));

TMatrix kernel = {

{0.1, 0.1, 0.1},

{0.1, 0.2, 0.1},

{0.1, 0.1, 0.1}

};

int iterations = 1;

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

ApplyConvolution(largeMatrix, kernel, iterations, 1); // Однопоточная версия

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

auto singleThreadTime = std::chrono::duration\_cast<std::chrono::milliseconds>(endSingle - startSingle).count();

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

ApplyConvolution(largeMatrix, kernel, iterations, 10); // Многопоточная версия

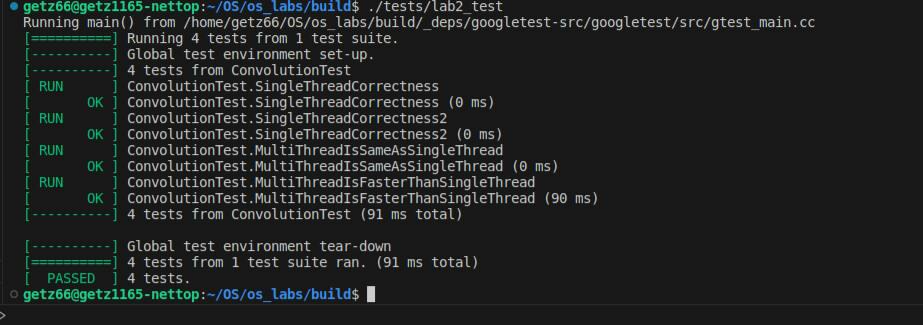
auto endMulti = std::chrono::high\_resolution\_clock::now();

auto multiThreadTime = std::chrono::duration\_cast<std::chrono::milliseconds>(endMulti - startMulti).count();

EXPECT\_GT(singleThreadTime, multiThreadTime);

}

**Демонстрация работы программы**

**Выводы**

Многопоточность является эффективным способом ускорения обработки данных. В данной задаче каждый поток обрабатывает свою часть матрицы, что значительно ускоряет выполнение программы по сравнению с однопоточной версией. Тесты с использованием Google Test показали корректность результатов многопоточной реализации, а также ускорение при увеличении числа потоков, особенно для больших матриц.

Создание потоков требует меньше ресурсов, чем создание процессов, и все потоки работают в одной области памяти. Таким образом, выполнение однотипных, не зависящих друг от друга задач, можно поручить отдельным потокам, которые будут работать параллельно. Это делает многопоточность удобным инструментом для задач с разделением данных.