Федеральное государственное бюджетное образовательное учреждение высшего образования

Национально исследовательский университет «Московский энергетический институт»

Кафедра ВМСС

Лабораторная работа №4

по курсу

«Программное обеспечение высокопроизводительных вычислительных систем»

Выполнили: студенты группы А-07м-23

Балашов С.А., Кретов Н.В., Михайлов С.А.

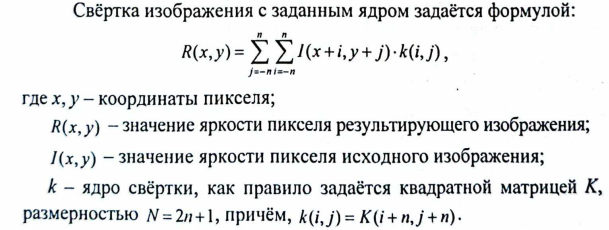
Проверил: Орлов Д.А., Хиль С.

Москва, 2024 г.

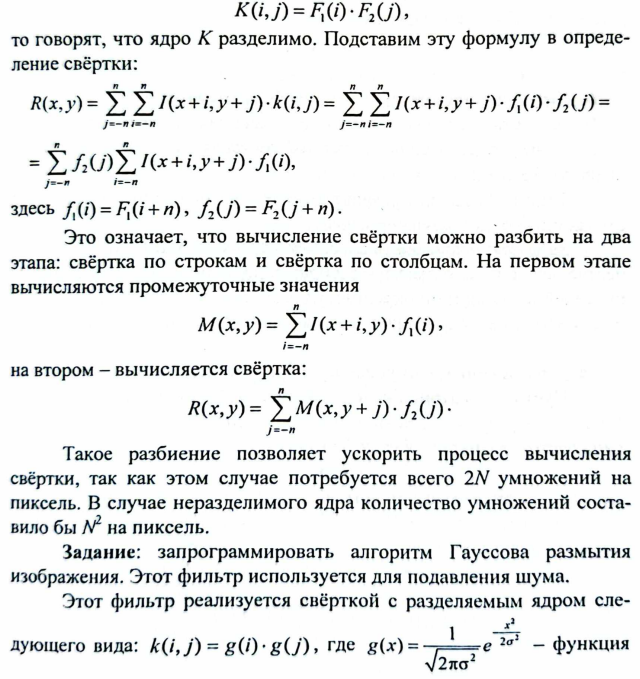
**Задание**

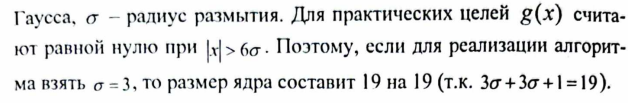
**Вариант 2**

**Свёртка с разделяемым ядром**



Если матрицу К можно представить в следующем виде:





Код программы:

#include <iostream>

#include <fstream>

#include <vector>

#include <opencv2/opencv.hpp>

using namespace std;

using namespace cv;

using namespace std::chrono;

#define PI 3.141592653

#define radSigCoeff 3

#define BLOCK\_SIZE 32

// Запрограммировать алгоритм Гауссова размытия. Этот фильтр используется для подавления шума.

#define checkCUDA(expression) { gpuAssert((expression), \_\_FILE\_\_, \_\_LINE\_\_); }

inline void gpuAssert(cudaError\_t status, const char\* file, int line)

{

if (status != cudaSuccess)

{

std::cerr << "GPUassert: " << cudaGetErrorString(status) << ", " << file << ", " << line << std::endl;

std::exit(EXIT\_FAILURE);

}

}

// Функция для расчёта времени выполнения функции

template<typename Func>

double measureExecutionTime(Func func) {

auto startTime = high\_resolution\_clock::now();

func();

auto endTime = high\_resolution\_clock::now();

auto duration = duration\_cast<nanoseconds>(endTime - startTime).count();

return duration / 1e6; // Перевод в миллисекунды

}

// Фильтр Гаусса 1D

void calculateKernel(float \*kernel, int sigma) {

int radius = radSigCoeff \* sigma;

for (int i = 0; i <= radius; i++) {

kernel[i + radius] = kernel[radius - i] = exp(

-(pow(i, 2) / (2 \* pow(sigma,2)))

) / (

sqrt(2 \* PI \* pow(sigma, 2))

);

}

}

\_\_global\_\_ void gaussianBlurRowsGPU(

const unsigned char \*srcImage, unsigned char \*dstImage, int width, int height,

const float \*kernel, int kernel\_radius) {

int col = blockIdx.x \* blockDim.x + threadIdx.x;

int row = blockIdx.y \* blockDim.y + threadIdx.y;

if ((row >= 0) && (col >= 0) && (row < height) && (col < width))

{

float sum = 0.0f;

float coeffSum = 0.0f;

float coeff = 0.0f;

for (int i = -kernel\_radius; i <= kernel\_radius; i++) {

coeff = kernel[i + kernel\_radius];

int currentCol = col + i;

if ((row \* width + currentCol) >= 0 && (row \* width + currentCol) < (width \* height)) {

sum += srcImage[row \* width + currentCol] \* coeff;

coeffSum += coeff;

}

}

dstImage[row \* width + col] = sum / (coeffSum > 0 ? coeffSum : 1);

}

}

\_\_global\_\_ void gaussianBlurColsGPU(

const unsigned char \*srcImage, unsigned char \*dstImage, int width, int height,

const float \*kernel, int kernel\_radius) {

int col = blockIdx.x \* blockDim.x + threadIdx.x;

int row = blockIdx.y \* blockDim.y + threadIdx.y;

if ((row >= 0) && (col >= 0) && (row < height) && (col < width))

{

float sum = 0.0f;

float coeffSum = 0.0f;

float coeff = 0.0f;

for (int i = -kernel\_radius; i <= kernel\_radius; i++) {

coeff = kernel[i + kernel\_radius];

int currentRow = row + i;

if ((col \* height + currentRow) >= 0 && (col \* height + currentRow) < (width \* height)) {

sum += srcImage[col \* height + currentRow] \* coeff;

coeffSum += coeff;

}

}

dstImage[col \* height + row] = sum / (coeffSum > 0 ? coeffSum : 1);

}

}

void runCuda(Mat &srcImage, Mat &dstImage, float \*kernel, int &kernel\_radius) {

cudaEvent\_t start, stop, mem1, mem2, mem3;

cudaEventCreate(&start);

cudaEventCreate(&stop);

cudaEventCreate(&mem1);

cudaEventCreate(&mem2);

cudaEventCreate(&mem3);

// Размеры изображения

int height = srcImage.rows;

int width = srcImage.cols;

const long long imageSize = height \* width;

unsigned char\* d\_input = nullptr, \* d\_output = nullptr, \* d\_tmp =nullptr;

float\* d\_kernel = nullptr;

Mat tmpImage = srcImage.clone();

cudaEventRecord(mem1);

// Выделение видеопамяти

checkCUDA(cudaMalloc(&d\_input, imageSize \* sizeof(unsigned char)));

checkCUDA(cudaMalloc(&d\_tmp, imageSize \* sizeof(unsigned char)));

checkCUDA(cudaMalloc(&d\_output, imageSize \* sizeof(unsigned char)));

checkCUDA(cudaMalloc(&d\_kernel, (kernel\_radius \* 2 + 1) \* sizeof(float)));

cudaEventRecord(mem2);

// Копирование изображения и фильтра в видеопамять

checkCUDA(cudaMemcpy(d\_input, srcImage.ptr(), imageSize \* sizeof(unsigned char), cudaMemcpyHostToDevice));

checkCUDA(cudaMemcpy(d\_kernel, kernel, (kernel\_radius \* 2 + 1) \* sizeof(float), cudaMemcpyHostToDevice));

// Размеры блока и сетки

dim3 block(BLOCK\_SIZE, BLOCK\_SIZE);

dim3 grid((int)ceil((float)width / block.x), (int)ceil((float)srcImage.rows / block.y));

// Запуск отсчёта

cudaEventRecord(start);

// Запуск ядер

gaussianBlurRowsGPU<<<grid, block>>>(d\_input, d\_tmp, width, height, d\_kernel, kernel\_radius);

gaussianBlurColsGPU<<<grid, block>>>(d\_tmp, d\_output, width, height, d\_kernel, kernel\_radius);

checkCUDA(cudaPeekAtLastError());

cudaEventRecord(mem3);

//Копирование результата из видеопамяти

checkCUDA(cudaMemcpy(dstImage.ptr(), d\_output, imageSize \* sizeof(unsigned char), cudaMemcpyDeviceToHost));

// Конец отсчёта

cudaEventRecord(stop);

//Очистка видеопамяти

cudaFree(d\_input);

cudaFree(d\_tmp);

cudaFree(d\_output);

cudaFree(d\_kernel);

cudaEventSynchronize(stop);

float time, mem1Time, mem2Time; // Время

cudaEventElapsedTime(&time, start, stop); // Расчёт времени

cudaEventElapsedTime(&mem1Time, mem1, mem2); // Расчёт времени

cudaEventElapsedTime(&mem2Time, mem3, stop); // Расчёт времени

ofstream globalTime("../gpu.csv", std::ios::app); // Открытие файла

cout << "\nProcessing time for shared memory GPU (ms): " << time << "\n"; // Вывод в поток

// Вывод в файл

globalTime << kernel\_radius << "," << width << "," << height << "," << time << "," << mem1Time << "," << mem2Time << "\n";

globalTime.close(); // Закрытие файла

}

void gaussianBlurRowsCPU(unsigned char \*srcImage, unsigned char \*dstImage, int width, int height,

float \*kernel, int kernel\_radius)

{

for (int y = 0; y < height; y++)

for (int x = 0; x < width; x++) {

float sum = 0;

for (int k = -kernel\_radius; k <= kernel\_radius; k++) {

int d = x + k;

if (d >= 0 && d < width)

sum += srcImage[y \* width + d] \* kernel[kernel\_radius - k];

}

dstImage[y \* width + x] = sum;

}

}

void gaussianBlurColsCPU(unsigned char \*srcImage, unsigned char \*dstImage, int width, int height,

float \*kernel, int kernel\_radius)

{

for (int y = 0; y < height; y++)

for (int x = 0; x < width; x++) {

float sum = 0;

for (int k = -kernel\_radius; k <= kernel\_radius; k++) {

int d = y + k;

if (d >= 0 && d < height)

sum += srcImage[d \* width + x] \* kernel[kernel\_radius - k];

}

dstImage[y \* width + x] = sum;

}

}

void runCpu(Mat &srcImage, Mat &dstImage, float \*kernel, int &kernel\_radius) {

Mat tmpImage = srcImage.clone();

gaussianBlurRowsCPU(srcImage.ptr(), tmpImage.ptr(), srcImage.cols, srcImage.rows, kernel, kernel\_radius);

gaussianBlurColsCPU(tmpImage.ptr(), dstImage.ptr(), srcImage.cols, srcImage.rows, kernel, kernel\_radius);

}

int main() {

vector<int> sigmas = {1, 2, 3, 4, 5};

vector<int> radii = {sigmas[0]\*radSigCoeff, sigmas[1]\*radSigCoeff, sigmas[2]\*radSigCoeff, sigmas[3]\*radSigCoeff, sigmas[4]\*radSigCoeff};

vector<float\*> kernels;

for (int i = 0; i < sigmas.size(); i++) {

auto \*kernel = new float[radii[i]\*2 + 1];

**calculateKernel(kernel, sigmas[i])**;

kernels.push\_back(kernel);

}

vector<string> paths = {

"image1.tiff",

"image2.tiff",

"image3.tiff",

};

for (const auto & path : paths)

{

Mat srcImage = imread("../"+path, IMREAD\_GRAYSCALE);

if (srcImage.empty()) {

cerr << "Could not open or find the image" << endl;

return -1;

}

Mat dstImage = srcImage.clone();

Mat dstImageCPU = srcImage.clone();

ofstream globalTime("../gpu.csv", std::ios::app);

globalTime << "radius" << "," << "width" << "," << "height" << "," << "time(ms)" << "," << "mem1(ms)" << "," << "mem2(ms)" << "\n";

globalTime.close();

ofstream cpuTime("../cpu.csv", std::ios::app);

cpuTime << "radius" << "," << "width" << "," << "height" << "," << "time(ms)" << "\n";

cpuTime.close();

double time = 0;

for (int i = 0; i < kernels.size(); i++) {

time = measureExecutionTime([&](){ **runCpu(srcImage, dstImageCPU, kernels[i], radii[i])**; });

ofstream cpuTime("../cpu.csv", std::ios::app);

cout << "\nProcessing time for CPU (ms): " << time << "\n";

cpuTime << radii[i] << "," << srcImage.cols << "," << srcImage.rows << "," << time << "\n";

cpuTime.close();

string output\_cpu = "../cpu\_" + to\_string(radii[i]) + "\_" + path;

imwrite(output\_cpu, dstImageCPU);

**runCuda(srcImage, dstImage, kernels[i], radii[i])**;

string output\_gpu = "../gpu\_" + to\_string(radii[i]) + "\_" + path;

imwrite(output\_gpu, dstImage);

}

}

for (auto & kernel : kernels) {

delete[] kernel;

}

return 0;

}

Время выполнения на CPU:

|  |  |  |  |
| --- | --- | --- | --- |
| **Радиус ядра** | **Ширина** | **Высота** | **Время выполнения, мс** |
| 3 | 256 | 256 | 2.7624 |
| 6 | 256 | 256 | 4.8505 |
| 9 | 256 | 256 | 6.9497 |
| 12 | 256 | 256 | 9.0927 |
| 15 | 256 | 256 | 11.3305 |
| 3 | 512 | 512 | 10.7474 |
| 6 | 512 | 512 | 19.3319 |
| 9 | 512 | 512 | 27.756 |
| 12 | 512 | 512 | 36.5825 |
| 15 | 512 | 512 | 45.2163 |
| 3 | 1024 | 1024 | 43.5986 |
| 6 | 1024 | 1024 | 77.0684 |
| 9 | 1024 | 1024 | 116.286 |
| 12 | 1024 | 1024 | 146.369 |
| 15 | 1024 | 1024 | 181.878 |

Время выполнения на GPU:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Радиус ядра** | **Ширина** | **Высота** | **Время выполнения, мс** | **Время обмена данными при решении задачи, мс** |
| 3 | 256 | 256 | 12.1537 | 0.424288 |
| 6 | 256 | 256 | 0.134048 | 0.434176 |
| 9 | 256 | 256 | 0.163072 | 0.402688 |
| 12 | 256 | 256 | 0.192256 | 0.382016 |
| 15 | 256 | 256 | 0.21104 | 0.374912 |
| 3 | 512 | 512 | 0.244672 | 0.750368 |
| 6 | 512 | 512 | 0.3536 | 0.514048 |
| 9 | 512 | 512 | 0.416096 | 0.505024 |
| 12 | 512 | 512 | 0.51312 | 0.520704 |
| 15 | 512 | 512 | 0.614496 | 0.521280 |
| 3 | 1024 | 1024 | 0.802944 | 0.775712 |
| 6 | 1024 | 1024 | 1.14717 | 0.769728 |
| 9 | 1024 | 1024 | 1.47283 | 0.923776 |
| 12 | 1024 | 1024 | 1.82765 | 0.776192 |
| 15 | 1024 | 1024 | 2.15974 | 0.792512 |

Коэффициент ускорения:

|  |  |  |  |
| --- | --- | --- | --- |
| **Радиус ядра** | **Ширина** | **Высота** | **Коэффициент ускорения выполнения программы на GPU относительно CPU** |
| 3 | 256 | 256 | 0.227289 |
| 6 | 256 | 256 | 36.184799 |
| 9 | 256 | 256 | 42.617371 |
| 12 | 256 | 256 | 47.294753 |
| 15 | 256 | 256 | 53.688874 |
| 3 | 512 | 512 | 43.925745 |
| 6 | 512 | 512 | 54.671663 |
| 9 | 512 | 512 | 66.705760 |
| 12 | 512 | 512 | 71.294239 |
| 15 | 512 | 512 | 73.582741 |
| 3 | 1024 | 1024 | 54.298432 |
| 6 | 1024 | 1024 | 67.181324 |
| 9 | 1024 | 1024 | 78.954122 |
| 12 | 1024 | 1024 | 80.085903 |
| 15 | 1024 | 1024 | 84.212915 |