**SVEUČILIŠTE U SPLITU**

**FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE**

NAPREDNE ARHITEKTURE RAČUNALA (250)

IZVJEŠTAJ 3

**Toni Biuk**

**Tino Melvan**

**Split, lipanj 2018.**

**SADRŽAJ**

[1. Specifikacije računala 1](#_Toc516516076)

[2. Gabor filter 2](#_Toc516516077)

[2.1. Kernel 2](#_Toc516516078)

[2.2. Implementacija 3](#_Toc516516079)

[2.3. Primjer obrade 4](#_Toc516516080)

[3. Implementacija konvolucije 5](#_Toc516516081)

[3.1. Kod 5](#_Toc516516082)

[3.2. Sekvencijalni rezultati 6](#_Toc516516083)

[3.3. Kod paralelne podijele 7](#_Toc516516084)

[3.4. Paralelni rezultati 7](#_Toc516516085)

[4. CUDA 10](#_Toc516516086)

[4.1. Kod konvolucije – global 10](#_Toc516516087)

[4.2. Kod konvolucije – host 10](#_Toc516516088)

[4.3. Rezultati 11](#_Toc516516089)

[5. Prilog 12](#_Toc516516090)

[5.1. inc/Filter/FilterProperties.hpp 12](#_Toc516516091)

[5.2. src/Filter/FilterProperties.cpp 12](#_Toc516516092)

[5.3. inc/Filter/GaborFilter.hpp 12](#_Toc516516093)

[5.4. src/Filter/GaborFilter.cpp 13](#_Toc516516094)

[5.5. inc/Image/Image.hpp 15](#_Toc516516095)

[5.6. src/Image/Image.cpp 15](#_Toc516516096)

[5.7. inc/Image/Convolution.hpp 16](#_Toc516516097)

[5.8. src/Image/Convolution.cpp 17](#_Toc516516098)

[5.9. src/Image/Convolution.cu 18](#_Toc516516099)

[5.10. inc/Utility/Stopwatch.hpp 20](#_Toc516516100)

[5.11. src/Utility/Stopwatch.cpp 21](#_Toc516516101)

[5.12. src/main.cpp 22](#_Toc516516102)

# Specifikacije računala

U ovom poglavlju navedene su relevantne specifikacije korištenog računala na kojima su se izvodili programi navedeni u ostalim poglavljima ili dani u nastavku. Specifikacije se nalaze u tablici ispod.

Tablica 1. Specifikacije računala

|  |  |
| --- | --- |
| Operacijski sustav | Microsoft Windows 10 Education Build 16299.309 |
| Procesor | Intel® Core™ i7-7700HQ @ 2.80GHz |
| Broj jezgri procesora | 4 |
| Broj niti procesora | 8 |
| RAM | 16GB (2X8GB) DDR4 – 2400MHz |
| Grafička kartica | GeForce GTX 1050 Ti |
| Broj CUDA jezgri | 768 |

# Gabor filter

Gabor filter jest linearni filter često korišten u digitalnoj obradi slike. Tu se primarno misli na analizu tekstura i detekciju rubova. Unutar slike, filter detektira različite frekvencije s njihovim odgovarajućim specifičnim smjerom. Slika se potom prostorno i frekvencijski lokalizira. U prostornoj domeni, ovaj filter predstavlja Gaussovu funkciju modeliranu matematičkom funkcijom sinusa.

## Kernel

Kod obrade slika, neizostavna stavka je kernel ili konvolucijska matrica. Ona se koristi za obavljanje transformacija nad slikama. Pod transformacijama misli se primjerice na izoštravanje, zamagljivanje, ili pak na detekciju rubova.

U osnovi, kako je već spomenuto, kernel je matrica (najčešće dvodimenzionalna) s unaprijed izračunatim vrijednostima. Te vrijednosti se potom primjenjuju na područje oko svakog piksela slike, i to konvolucijom. Na taj način dobiva se nova vrijednost svakog piksela. Same vrijednosti matrice kernela izračunavaju se prema sljedećoj formuli:



gdje su:





Koeficijenti redom predstavljaju:

λ – valna dužina sinusoidalnog faktora

θ – orijentacija normala paralelnih pruga Gaborovog filtera

ψ – fazni pomak

σ – standardna devijacija Gausove omotnice

γ – omjer prostornog aspekta

## Implementacija

Implementacija dane formule u C++ programskom jeziku nalazi se u nastavku. Također, bitno je napomenuti da je za izradu ovog seminarskog rada korištena otvorena biblioteka OpenCV.

cv::Mat GaborFilter::GetGaborKernel(const cv::Size &size,

const double sigma,

const double theta,

const double lambda,

const double gamma,

const double psi) const {

if (size.empty()) {

return cv::Mat(1, 1, CV\_32F, cv::Scalar(1, 0, 0));

}

assert(sigma != 0.0);

assert(lambda != 0.0);

assert(gamma != 0.0);

const auto rowMax = size.height / 2;

const auto columnMax = size.width / 2;

const auto rowMin = -rowMax;

const auto columnMin = -columnMax;

const auto sigmaRow = -0.5 / std::pow(sigma / gamma, 2);

const auto sigmaColumn = -0.5 / std::pow(sigma, 2);

cv::Mat kernel(rowMax - rowMin + 1, columnMax - columnMin + 1, CV\_32F);

for (auto row = rowMin; row <= rowMax; row++) {

for (auto column = columnMin; column <= columnMax; column++) {

const auto thetaRow = -column \* sin(theta) + row \* cos(theta);

const auto thetaColumn = column \* cos(theta) + row \* sin(theta);

const auto gabor = std::exp(sigmaColumn \* thetaColumn \* thetaColumn +

sigmaRow \* thetaRow \* thetaRow) \*

std::cos(2 \* PI \* thetaColumn / lambda + psi);

kernel.at<float>(rowMax - row, columnMax - column) =

static\_cast<float>(gabor);

}

}

return kernel;

}

## Primjer obrade

Izračunata vrijednost svakog pojedinog piksela ovisi isključivo o vrijednostima kernela, kao i o vrijednostima okolnih piksela originalne slike. Ova činjenica omogućava jednostavnu paralelizaciju o kojoj će biti više riječi u narednim poglavljima. Uzmimo za primjer sliku 2.1. Ukoliko na nju primjenimo Gaborov kernel s ulaznim parametrima λ = 6.0; θ = 1.0; ψ = 1.0; σ = 3.0; γ = 6.5, dobiti ćemo sliku 2.2.



Slika 2.1. Neobrađena slika



Slika 2.2. Obrađena slika

Na obrađenoj slici se jasno mogu razlučiti razne teksture, poput oblaka, neba ili mora.

# Implementacija konvolucije

## Kod

cv::Mat Convolve(const cv::Mat& image, const cv::Mat& kernel, struct Thread thread) {

auto output = image.clone();

if (thread.step == NO\_THREAD\_STEP) {

thread.step = image.rows;

}

const auto kernelCenterRow = (kernel.rows - 1) / 2;

const auto kernelCenterColumn = (kernel.cols - 1) / 2;

for (auto imageRow = thread.id \* thread.step;

imageRow < (thread.id + 1) \* thread.step;

imageRow++) {

if (imageRow >= image.rows) {

break;

}

for (auto imageColumn = 0; imageColumn < image.cols; imageColumn++) {

auto sum = 0.0F;

for (auto kernelRow = -kernelCenterRow;

kernelRow <= kernelCenterRow;

kernelRow++) {

for (auto kernelColumn = -kernelCenterColumn;

kernelColumn <= kernelCenterColumn;

kernelColumn++) {

if (imageRow + kernelRow <= 0 ||

imageRow + kernelRow >= image.rows ||

imageColumn + kernelColumn <= 0 ||

imageColumn + kernelColumn >= image.cols) {

continue;

}

sum += kernel.at<float>(kernelRow + kernelCenterRow,

kernelColumn + kernelCenterColumn) \*

image.at<float>(imageRow + kernelRow,

imageColumn + kernelColumn);

}

}

output.at<float>(imageRow, imageColumn) = sum;

}

}

if ((thread.id + 1) \* thread.step > image.rows) {

return output.rowRange(thread.id \* thread.step, image.rows);

}

return output.rowRange(thread.id \* thread.step, (thread.id + 1) \* thread.step);

}

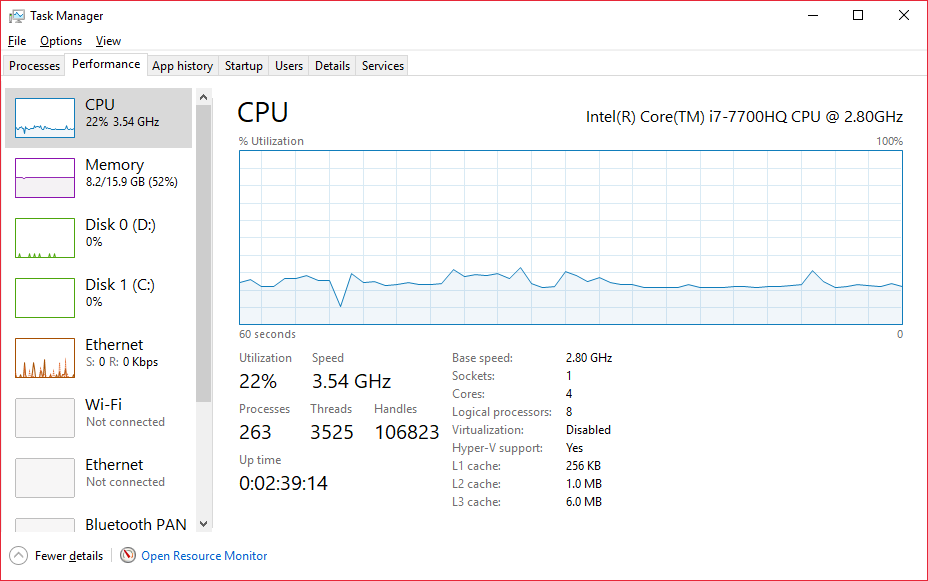
## Sekvencijalni rezultati

Funkcija prima tri parametra: matricu crno-bijele ulazne slike, matricu kernela i strukturu kojom je definirano hoće li funkcija biti izvedena sekvencijalno ili paralelno. U slučaju paralelnog izvođenja, istom strukturom se definira broj niti.

Slika 3.1 prikazuje sekvencijalno vrijeme izvođenja programa ovisno o veličini slike. Za konvoluciju slika se koristi kernel veličine 11x11. Vidi se linearna ovisnost vremena izvođenja i broja piksela slike. Na primjer, slika veličine 2560x1920 sadrži približno 600 milijuna piksela, dok slika veličine 4096x3072 ima oko 1.5 milijarde što je približno tri puta više, a iz grafa se vidi da je vrijeme izvođenja za drugu sliku približno tri puta duže.

Slika 3.1. Graf sekvencijalnog vremena izvođenja programa

Slika 3.2 prikazuje performanse procesora prilikom sekvencijalnog izvođenja programa. Vidljivo je da program koristi samo oko 20% dostupne snage procesora jer je prilikom sekvencijalnog izvođenja programa moguće korištenje samo jedne od osam dostupnih niti. Također vrijedi naglasiti da je zbog Intel-ove Turbo Boost Tehnologije frekvencija povećana do 3.8 GHz kada se koristi samo jedna jezgra.



Slika 3.2. Performanse sekvencijalnog izvođenja

## Kod paralelne podijele

cv::Mat convolution::Parallel(const cv::Mat& image, const cv::Mat& kernel) {

cv::Mat output;

const auto numberOfThreads = std::max(cv::getNumThreads(),

cv::getNumberOfCPUs()) - 1;

const auto splitRowCount = static\_cast<int>

(std::ceil(static\_cast<double>(image.rows) / numberOfThreads));

std::vector<std::future<cv::Mat>> threads;

for (auto i = 0; i < numberOfThreads; i++) {

const struct Thread thread = { i, splitRowCount };

threads.emplace\_back(std::async(Convolve, image, kernel, thread));

}

for (auto &thread : threads) {

const auto processedImagePart = thread.get();

output.push\_back(processedImagePart);

}

return output;

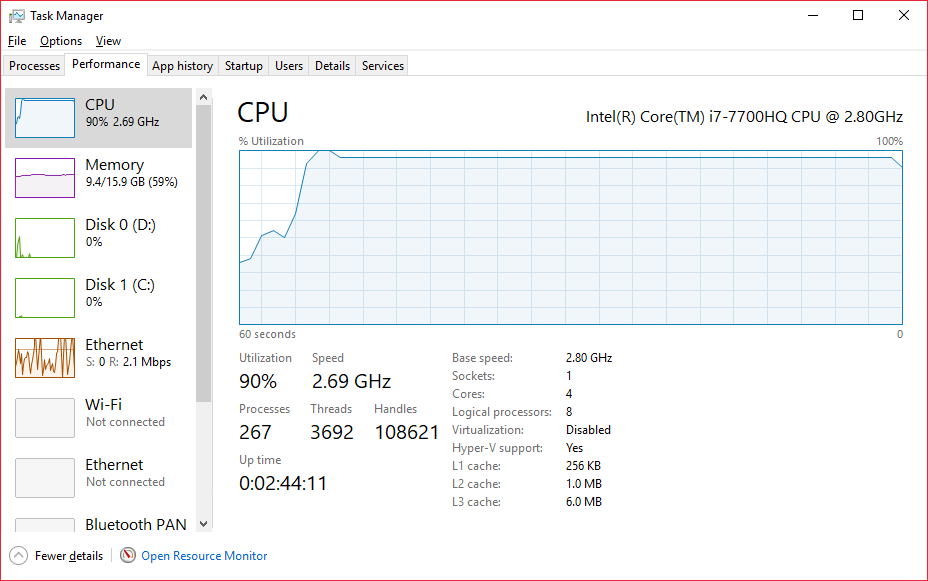
}

## Paralelni rezultati

Za paralelno izvođenje programa se koristi ista funkcija za konvoluciju kao i kod sekvencijalne izvedbe uz promijenjene parametre koji definiraju postavke za paralelno izvođenje programa. Rezultati izvođenja su prikazani na slici 3.3. Odmah se primjećuje identična linearna ovisnost vremena izvođenja i vremena izvođenja programa kao kod sekvencijalnog izvođenja.

Slika 3.3. Paralelno vrijeme izvođenja programa

Također je vidljivo ubrzanje od 4 do 5 puta u odnosu na sekvencijalno izvođenje. Za razliku od sekvencijalnog izvođenja kod koje je korištena samo jedna od osam dostupnih niti procesora, paralelna izvedba koristi svih osam. Logično bi bilo pretpostaviti da korištenje osam puta više niti rezultira osam puta bržim izvođenjem, ali u praksi se to ne događa najvećim dijelom zbog prije spomenute Intel Turbo Boost Tehnologije koja omogućuje povećanje frekvencije jezgre kada se koristi samo jedna jezgra.



Slika 3.4. Performanse paralelnog izvođenja

Za razliku od sekvencijalnog izvođenja, paralelno izvođenje maksimalno iskorištava dostupne resurse procesora što se poklapa sa zapažanjima o bržem vremenu izvođenja.

# CUDA

## Kod konvolucije – global

\_\_global\_\_

void CudaConvolve(const cv::cuda::PtrStepSz<float> image,

cv::cuda::PtrStepSz<float> output,

const cv::cuda::PtrStepSz<float> kernel) {

const auto pixelRow = threadIdx.y + blockIdx.y \* blockDim.y;

const auto pixelColumn = threadIdx.x + blockIdx.x \* blockDim.x;

if (pixelRow >= image.rows || pixelColumn >= image.cols) {

return;

}

const auto kernelCenterRow = (kernel.rows - 1) / 2;

const auto kernelCenterColumn = (kernel.cols - 1) / 2;

auto sum = 0.0F;

for (auto kernelRow = -kernelCenterRow;

kernelRow <= kernelCenterRow;

kernelRow++) {

for (auto kernelColumn = -kernelCenterColumn;

kernelColumn <= kernelCenterColumn;

kernelColumn++) {

if (pixelRow + kernelRow <= 0 ||

pixelRow + kernelRow >= image.rows ||

pixelColumn + kernelColumn <= 0 ||

pixelColumn + kernelColumn >= image.cols) {

continue;

}

sum += kernel.ptr(kernelRow + kernelCenterRow)[kernelColumn + kernelCenterColumn] \* image.ptr(pixelRow + kernelRow)[pixelColumn + kernelColumn];

}

}

output.ptr(pixelRow)[pixelColumn] = sum;

}

## Kod konvolucije – host

cv::Mat convolution::Cuda(const cv::Mat &image, const cv::Mat &kernel) {

cv::cuda::GpuMat output\_d(image.clone());

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

dim3 numBlocks(static\_cast<uint32\_t>

(std::ceil(static\_cast<float>(image.cols) / threadsPerBlock.x)),

static\_cast<uint32\_t>

(std::ceil(static\_cast<float>(image.rows) / threadsPerBlock.y)));

CudaConvolve<<<numBlocks, threadsPerBlock>>>(cv::cuda::GpuMat(image),

output\_d,

cv::cuda::GpuMat(kernel));

GPU\_ERROR\_CHECK(cudaPeekAtLastError());

GPU\_ERROR\_CHECK(cudaDeviceSynchronize());

cv::Mat output;

output\_d.download(output);

output\_d.release();

return output;

}

## Rezultati

Kod paralelne implementacije slika se podijeli više jednakih dijelova (po redovima). Svaki dio se potom pošalje u jednu nit na obradu. Takva implementacija, pak, nije optimalna za rad na grafičkoj kartici. CUDA ulazni podatak dijeli na blokove (u radu uzeti blokovi 16x16) i te blokove obrađuje. S obzirom na to da je naš ulazni podatak slika, bolja opcija je podijeliti tu sliku na manje blokove koje ćemo potom poslati na grafičku karticu. CUDA ima mnogo jezgri što je idealno za obradu takve vrste zato što, do slika određenih veličina (ovisi o grafičkoj kartici), svaki piksel može dobiti svoju nit za obradu.

Ovisnost vremena o veličini izvođenja nalazi se na slici 4.1. Uočljive su velike razlike u usporedbi s ostale dvije implementacije. Za razliku od sekvencijalne i paralelne implementacije, CUDA zahtijeva mnogo vremena za pokretanje i kopiranje podataka na grafičku karticu. Zbog tog dugog vremena pripreme, vrijeme izvršavanja za najmanju korištenu sliku je čak sporije nego kod paralelne implementacije.

Iako su performanse loše kod malih veličina slika, prednosti su jako izrazite na velikim slikama. Najizrazitija je razlika između sekvencijalne i CUDA implementacije kod najveće korištenje slike gdje se sekvencijalni program izvršava više od tri minute dok je CUDA implementacija gotova u samo tri sekunde.

Slika 4.1. CUDA vrijeme izvršavanja programa

# Prilog

## inc/Filter/FilterProperties.hpp

#ifndef FILTER\_FILTERPROPERTIES\_HPP

#define FILTER\_FILTERPROPERTIES\_HPP

struct FilterProperties {

double deviation;

double orientation;

double wavelength;

double ratio;

double offset;

FilterProperties() noexcept;

};

#endif // !FILTER\_FILTERPROPERTIES\_HPP

## src/Filter/FilterProperties.cpp

#include "Filter/FilterProperties.hpp"

#define DEFAULT\_WAVELENGTH 1.0 // λ - lambda

#define DEFAULT\_NORMAL\_ORIENTATION 0.0 // θ - theta

#define DEFAULT\_PHASE\_OFFSET 0.0 // ψ - psi

#define DEFAULT\_STANDARD\_DEVIATION (0.56 \* DEFAULT\_WAVELENGTH) // σ - sigma

#define DEFAULT\_SPATIAL\_ASPECT\_RATIO 0.02 // γ - gamma

FilterProperties::FilterProperties() noexcept {

deviation = DEFAULT\_STANDARD\_DEVIATION;

orientation = DEFAULT\_NORMAL\_ORIENTATION;

wavelength = DEFAULT\_WAVELENGTH;

ratio = DEFAULT\_SPATIAL\_ASPECT\_RATIO;

offset = DEFAULT\_PHASE\_OFFSET;

}

## inc/Filter/GaborFilter.hpp

#ifndef FILTER\_FILTER\_HPP

#define FILTER\_FILTER\_HPP

#include <Filter/FilterProperties.hpp>

#include <opencv2/core/mat.hpp>

#include <opencv2/core/types.hpp>

#include <cstdint>

struct GaborFilter {

GaborFilter();

GaborFilter(int32\_t kernelRowsSize, int32\_t kernelColumnsSize);

explicit GaborFilter(const cv::Size &size);

void SetDeviation(double deviation) noexcept;

void SetOrientation(double orientation) noexcept;

void SetWaveLength(double wavelength) noexcept;

void SetSpatialAspectRatio(double ratio) noexcept;

void SetPhaseOffset(double offset) noexcept;

void AdjustWithBandwith(double bandwith) noexcept;

void RefreshKernel();

cv::Mat kernel;

private:

cv::Mat GetGaborKernel(const cv::Size &size,

double sigma,

double theta,

double lambda,

double gamma,

double psi) const;

cv::Size m\_Size;

FilterProperties m\_Properties;

};

#endif // !FILTER\_FILTER\_HPP

## src/Filter/GaborFilter.cpp

#include "Filter/GaborFilter.hpp"

#include <opencv2/core/hal/interface.h>

#include <opencv2/imgproc/imgproc.hpp>

#include <opencv2/core/types.hpp>

#include <cmath>

#define DEFAULT\_KERNEL\_SIZE 3U

#define PI 3.1415926535897932384626433832795

GaborFilter::GaborFilter() {

this->m\_Size.height = DEFAULT\_KERNEL\_SIZE;

this->m\_Size.width = DEFAULT\_KERNEL\_SIZE;

RefreshKernel();

}

GaborFilter::GaborFilter(const cv::Size &size) {

this->m\_Size = size;

RefreshKernel();

}

GaborFilter::GaborFilter(const int32\_t kernelRowsSize,

const int32\_t kernelColumnsSize) {

this->m\_Size.height = kernelRowsSize;

this->m\_Size.width = kernelColumnsSize;

RefreshKernel();

}

void GaborFilter::SetDeviation(const double deviation) noexcept {

this->m\_Properties.deviation = deviation;

}

void GaborFilter::SetOrientation(const double orientation) noexcept {

this->m\_Properties.deviation = orientation;

}

void GaborFilter::SetWaveLength(const double wavelength) noexcept {

this->m\_Properties.deviation = wavelength;

}

void GaborFilter::SetSpatialAspectRatio(const double ratio) noexcept {

this->m\_Properties.deviation = ratio;

}

void GaborFilter::SetPhaseOffset(const double offset) noexcept {

this->m\_Properties.deviation = offset;

}

void GaborFilter::AdjustWithBandwith(const double bandwith) noexcept {

this->m\_Properties.deviation = this->m\_Properties.wavelength;

this->m\_Properties.deviation /= PI;

this->m\_Properties.deviation \*= std::sqrt(std::log(2.0) / 2);

this->m\_Properties.deviation \*= std::pow(2, bandwith) + 1;

this->m\_Properties.deviation /= std::pow(2, bandwith) - 1;

}

void GaborFilter::RefreshKernel() {

this->kernel = GetGaborKernel(this->m\_Size,

this->m\_Properties.deviation,

this->m\_Properties.orientation,

this->m\_Properties.wavelength,

this->m\_Properties.ratio,

this->m\_Properties.offset);

}

cv::Mat GaborFilter::GetGaborKernel(const cv::Size &size,

const double sigma,

const double theta,

const double lambda,

const double gamma,

const double psi) const {

if (size.empty()) {

return cv::Mat(1, 1, CV\_32F, cv::Scalar(1, 0, 0));

}

assert(sigma != 0.0);

assert(lambda != 0.0);

assert(gamma != 0.0);

const auto rowMax = size.height / 2;

const auto columnMax = size.width / 2;

const auto rowMin = -rowMax;

const auto columnMin = -columnMax;

const auto sigmaRow = -0.5 / std::pow(sigma / gamma, 2);

const auto sigmaColumn = -0.5 / std::pow(sigma, 2);

cv::Mat kernel(rowMax - rowMin + 1, columnMax - columnMin + 1, CV\_32F);

for (auto row = rowMin; row <= rowMax; row++) {

for (auto column = columnMin; column <= columnMax; column++) {

const auto thetaRow = -column \* sin(theta) + row \* cos(theta);

const auto thetaColumn = column \* cos(theta) + row \* sin(theta);

const auto gabor = std::exp(sigmaColumn \* thetaColumn \* thetaColumn +

sigmaRow \* thetaRow \* thetaRow) \*

std::cos(2 \* PI \* thetaColumn / lambda + psi);

kernel.at<float>(rowMax - row, columnMax - column) =

static\_cast<float>(gabor);

}

}

return kernel;

}

## inc/Image/Image.hpp

#ifndef IMAGE\_IMAGE\_HPP

#define IMAGE\_IMAGE\_HPP

#include <opencv2/core/mat.hpp>

#include <opencv2/imgcodecs.hpp>

#include <cstdint>

#include <string>

struct Image {

explicit Image(const cv::Mat &data);

explicit Image(const std::string &fileName);

Image(const std::string &fileName, int32\_t mode);

void ReadFromFile(const std::string &fileName,

int32\_t mode = cv::IMREAD\_COLOR);

void WriteToFile(const std::string &fileName) const;

cv::Mat Format(int32\_t format, double scaleFactor = 1.0) const;

void FormatItself(int32\_t format, double scaleFactor = 1.0);

cv::Mat GetData() const;

void SetData(const cv::Mat &data);

private:

cv::Mat m\_ImageData;

};

#endif // !IMAGE\_IMAGE\_HPP

## src/Image/Image.cpp

#include "Image/Image.hpp"

#include <opencv2/imgcodecs.hpp>

#include <opencv2/core/mat.hpp>

#include <cstdint>

#include <string>

Image::Image(const cv::Mat& data) {

this->m\_ImageData = data;

}

Image::Image(const std::string& fileName) {

this->m\_ImageData = cv::imread(fileName);

}

Image::Image(const std::string& fileName, const int32\_t mode) {

this->m\_ImageData = cv::imread(fileName, mode);

}

void Image::ReadFromFile(const std::string& fileName, const int32\_t mode) {

this->m\_ImageData = cv::imread(fileName, mode);

}

void Image::WriteToFile(const std::string& fileName) const {

cv::imwrite(fileName, this->m\_ImageData);

}

cv::Mat Image::Format(const int32\_t format, const double scaleFactor) const {

cv::Mat formattedImage;

this->m\_ImageData.convertTo(formattedImage, format, scaleFactor);

return formattedImage;

}

void Image::FormatItself(const int32\_t format, const double scaleFactor) {

cv::Mat formattedImage;

this->m\_ImageData.convertTo(formattedImage, format, scaleFactor);

this->m\_ImageData = formattedImage;

}

cv::Mat Image::GetData() const {

return this->m\_ImageData;

}

void Image::SetData(const cv::Mat &data) {

this->m\_ImageData = data;

}

## inc/Image/Convolution.hpp

#ifndef IMAGE\_CONVOLUTION\_HPP

#define IMAGE\_CONVOLUTION\_HPP

#include <opencv2/core/mat.hpp>

namespace convolution {

cv::Mat Sequential(const cv::Mat &image, const cv::Mat &kernel);

cv::Mat Parallel(const cv::Mat &image, const cv::Mat &kernel);

cv::Mat Cuda(const cv::Mat &image, const cv::Mat &kernel);

}

#endif // !IMAGE\_CONVOLUTION\_HPP

## src/Image/Convolution.cpp

#include "Image/Convolution.hpp"

#include <opencv2/core/mat.hpp>

#include <opencv2/core/utility.hpp>

#include <algorithm>

#include <future>

#include <vector>

#define NO\_THREAD 0

#define NO\_THREAD\_STEP (-1)

struct Thread {

int32\_t id;

int32\_t step;

};

cv::Mat Convolve(const cv::Mat& image, const cv::Mat& kernel, struct Thread thread) {

auto output = image.clone();

if (thread.step == NO\_THREAD\_STEP) {

thread.step = image.rows;

}

const auto kernelCenterRow = (kernel.rows - 1) / 2;

const auto kernelCenterColumn = (kernel.cols - 1) / 2;

for (auto imageRow = thread.id \* thread.step;

imageRow < (thread.id + 1) \* thread.step;

imageRow++) {

if (imageRow >= image.rows) {

break;

}

for (auto imageColumn = 0; imageColumn < image.cols; imageColumn++) {

auto sum = 0.0F;

for (auto kernelRow = -kernelCenterRow;

kernelRow <= kernelCenterRow;

kernelRow++) {

for (auto kernelColumn = -kernelCenterColumn;

kernelColumn <= kernelCenterColumn;

kernelColumn++) {

if (imageRow + kernelRow <= 0 ||

imageRow + kernelRow >= image.rows ||

imageColumn + kernelColumn <= 0 ||

imageColumn + kernelColumn >= image.cols) {

continue;

}

sum += kernel.at<float>(kernelRow + kernelCenterRow,

kernelColumn + kernelCenterColumn) \*

image.at<float>(imageRow + kernelRow,

imageColumn + kernelColumn);

}

}

output.at<float>(imageRow, imageColumn) = sum;

}

}

if ((thread.id + 1) \* thread.step > image.rows) {

return output.rowRange(thread.id \* thread.step, image.rows);

}

return output.rowRange(thread.id \* thread.step, (thread.id + 1) \* thread.step);

}

cv::Mat convolution::Sequential(const cv::Mat& image, const cv::Mat& kernel) {

return Convolve(image, kernel, { NO\_THREAD, NO\_THREAD\_STEP });

}

cv::Mat convolution::Parallel(const cv::Mat& image, const cv::Mat& kernel) {

cv::Mat output;

const auto numberOfThreads = std::max(cv::getNumThreads(),

cv::getNumberOfCPUs()) - 1;

const auto splitRowCount = static\_cast<int>

(std::ceil(static\_cast<double>(image.rows) / numberOfThreads));

std::vector<std::future<cv::Mat>> threads;

for (auto i = 0; i < numberOfThreads; i++) {

const struct Thread thread = { i, splitRowCount };

threads.emplace\_back(std::async(Convolve, image, kernel, thread));

}

for (auto &thread : threads) {

const auto processedImagePart = thread.get();

output.push\_back(processedImagePart);

}

return output;

}

## src/Image/Convolution.cu

#include "Image/Convolution.hpp"

#ifdef \_\_CUDACC\_\_

#include <cuda.h>

#include <cuda\_runtime.h>

#include <device\_launch\_parameters.h>

#include <opencv2/core/cuda.hpp>

#include <opencv2/core/cuda\_types.hpp>

#endif

#include <opencv2/core/mat.hpp>

#include <opencv2/core/types.hpp>

#include <cstdint>

#include <cstdio>

#ifdef \_\_CUDACC\_\_

#define BLOCK\_SIZE 16U

#define GPU\_ERROR\_CHECK(ans) { GpuAssert((ans), \_\_FILE\_\_, \_\_LINE\_\_); }

inline void GpuAssert(const cudaError\_t code,

const char \*file,

const int32\_t line) {

if (code == cudaSuccess) {

return;

}

fprintf(stderr,

"GpuAssert: %s %s %d\n",

cudaGetErrorString(code),

file,

line);

exit(code);

}

\_\_global\_\_

void CudaConvolve(const cv::cuda::PtrStepSz<float> image,

cv::cuda::PtrStepSz<float> output,

const cv::cuda::PtrStepSz<float> kernel) {

const auto pixelRow = threadIdx.y + blockIdx.y \* blockDim.y;

const auto pixelColumn = threadIdx.x + blockIdx.x \* blockDim.x;

if (pixelRow >= image.rows || pixelColumn >= image.cols) {

return;

}

const auto kernelCenterRow = (kernel.rows - 1) / 2;

const auto kernelCenterColumn = (kernel.cols - 1) / 2;

auto sum = 0.0F;

for (auto kernelRow = -kernelCenterRow;

kernelRow <= kernelCenterRow;

kernelRow++) {

for (auto kernelColumn = -kernelCenterColumn;

kernelColumn <= kernelCenterColumn;

kernelColumn++) {

if (pixelRow + kernelRow <= 0 ||

pixelRow + kernelRow >= image.rows ||

pixelColumn + kernelColumn <= 0 ||

pixelColumn + kernelColumn >= image.cols) {

continue;

}

sum += kernel.ptr(kernelRow + kernelCenterRow)[kernelColumn + kernelCenterColumn] \* image.ptr(pixelRow + kernelRow)[pixelColumn + kernelColumn];

}

}

output.ptr(pixelRow)[pixelColumn] = sum;

}

cv::Mat convolution::Cuda(const cv::Mat &image, const cv::Mat &kernel) {

cv::cuda::GpuMat output\_d(image.clone());

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

dim3 numBlocks(static\_cast<uint32\_t>

(std::ceil(static\_cast<float>(image.cols) / threadsPerBlock.x)),

static\_cast<uint32\_t>

(std::ceil(static\_cast<float>(image.rows) / threadsPerBlock.y)));

CudaConvolve<<<numBlocks, threadsPerBlock>>>(cv::cuda::GpuMat(image),

output\_d,

cv::cuda::GpuMat(kernel));

GPU\_ERROR\_CHECK(cudaPeekAtLastError());

GPU\_ERROR\_CHECK(cudaDeviceSynchronize());

cv::Mat output;

output\_d.download(output);

output\_d.release();

return output;

}

#else

#pragma message("CUDA-NOT-SUPPORTED!")

cv::Mat convolution::Cuda(const cv::Mat &image, const cv::Mat &kernel) {

return cv::Mat(image.rows, image.cols, CV\_32F, cv::Scalar(0, 0, 0));

}

#endif

## inc/Utility/Stopwatch.hpp

#ifndef UTILITY\_STOPWATCH\_HPP

#define UTILITY\_STOPWATCH\_HPP

#include <cstdint>

#include <chrono>

struct Stopwatch {

Stopwatch() noexcept;

void Start() noexcept;

void Pause() noexcept;

void Reset() noexcept;

void Restart() noexcept;

bool IsRunning() const noexcept;

int64\_t ElapsedNanoseconds() const noexcept;

int64\_t ElapsedMicroSeconds() const noexcept;

int64\_t ElapsedMiliseconds() const noexcept;

int64\_t ElapsedSeconds() const noexcept;

int64\_t ElapsedMinutes() const noexcept;

int64\_t ElapsedHours() const noexcept;

private:

std::chrono::duration<double> Elapsed() const noexcept;

bool m\_Running;

std::chrono::time\_point<std::chrono::system\_clock> m\_StartTime;

std::chrono::time\_point<std::chrono::system\_clock> m\_EndTime;

};

#endif // !UTILITY\_STOPWATCH\_HPP

## src/Utility/Stopwatch.cpp

#include "Utility/Stopwatch.hpp"

Stopwatch::Stopwatch() noexcept {

m\_Running = false;

m\_StartTime = std::chrono::system\_clock::now();

m\_EndTime = m\_StartTime;

}

void Stopwatch::Start() noexcept {

if (m\_StartTime == m\_EndTime) {

Restart();

} else {

m\_Running = true;

}

}

void Stopwatch::Pause() noexcept {

m\_Running = false;

m\_EndTime = std::chrono::system\_clock::now();

}

void Stopwatch::Reset() noexcept {

m\_Running = false;

m\_StartTime = std::chrono::system\_clock::now();

m\_EndTime = m\_StartTime;

}

void Stopwatch::Restart() noexcept {

m\_Running = true;

m\_StartTime = std::chrono::system\_clock::now();

m\_EndTime = m\_StartTime;

}

bool Stopwatch::IsRunning() const noexcept {

return m\_Running;

}

int64\_t Stopwatch::ElapsedNanoseconds() const noexcept {

return std::chrono::duration\_cast<std::chrono::nanoseconds>(Elapsed()).count();

}

int64\_t Stopwatch::ElapsedMicroSeconds() const noexcept {

return std::chrono::duration\_cast<std::chrono::microseconds>(Elapsed()).count();

}

int64\_t Stopwatch::ElapsedMiliseconds() const noexcept {

return std::chrono::duration\_cast<std::chrono::milliseconds>(Elapsed()).count();

}

int64\_t Stopwatch::ElapsedSeconds() const noexcept {

return std::chrono::duration\_cast<std::chrono::seconds>(Elapsed()).count();

}

int64\_t Stopwatch::ElapsedMinutes() const noexcept {

return std::chrono::duration\_cast<std::chrono::minutes>(Elapsed()).count();

}

int64\_t Stopwatch::ElapsedHours() const noexcept {

return std::chrono::duration\_cast<std::chrono::hours>(Elapsed()).count();

}

std::chrono::duration<double> Stopwatch::Elapsed() const noexcept {

if (IsRunning()) {

const auto now = std::chrono::system\_clock::now();

return now - m\_StartTime;

}

return m\_EndTime - m\_StartTime;

}

## src/main.cpp

#include <Filter/GaborFilter.hpp>

#include <Image/Convolution.hpp>

#include <Image/Image.hpp>

#include <Utility/Stopwatch.hpp>

#include <opencv2/core/mat.hpp>

#include <opencv2/core/hal/interface.h>

#include <cstdio>

#include <cstdint>

#include <string>

#define EXTRA\_SMALL "assets/s\_128x256.png"

#define SMALL "assets/800x600.jpg"

#define MEDIUM "assets/1280x960.jpg"

#define LARGE "assets/1920x1440.jpg"

#define EXTRA\_LARGE "assets/2560x1920.jpg"

#define EXTRA\_EXTRA\_LARGE "assets/4096x3072.jpg"

#define PATH\_TO\_IMAGE (MEDIUM)

void SavePng(const cv::Mat &image, const std::string &fileName) {

Image result(image);

result.FormatItself(CV\_8U, 1.0 / 255.0);

result.WriteToFile(fileName);

}

int32\_t main() {

Stopwatch stopwatch;

Image image(PATH\_TO\_IMAGE, cv::IMREAD\_GRAYSCALE);

image.FormatItself(CV\_32F);

GaborFilter filter(3, 3);

filter.RefreshKernel();

// Sequential

stopwatch.Start();

const auto convResultSequential = convolution::Sequential(image.GetData(),

filter.kernel);

stopwatch.Pause();

SavePng(convResultSequential, "resultSequential.png");

printf("Sequential:\t%lld ms\n", stopwatch.ElapsedMiliseconds());

// !Sequential

// Parallel

stopwatch.Restart();

const auto convResultParallel = convolution::Parallel(image.GetData(),

filter.kernel);

stopwatch.Pause();

SavePng(convResultParallel, "resultParallel.png");

printf("Parallel:\t%lld ms\n", stopwatch.ElapsedMiliseconds());

// !Parallel

// CUDA

stopwatch.Restart();

const auto convResultCuda = convolution::Cuda(image.GetData(),

filter.kernel);

stopwatch.Pause();

SavePng(convResultCuda, "resultCUDA.png");

printf("CUDA:\t\t%lld ms\n", stopwatch.ElapsedMiliseconds());

// !CUDA

return EXIT\_SUCCESS;

}