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

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

2. 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.

2.1. 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:

$$g(x, y; \lambda, \theta, \psi, \sigma, \gamma) = \exp\left(-\frac{x'^2 + \gamma^2 y'^2}{2\sigma^2}\right) \exp\left(i\left(2\pi\frac{x'}{\lambda} + \psi\right)\right)$$

gdje su:

$$x' = x\cos\theta + y\sin\theta$$

$$y' = -x\sin\theta + y\cos\theta$$

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

2.2. 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;
}
```

2.3. 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 $\lambda = 6.0$; $\theta = 1.0$; $\psi = 1.0$; $\sigma = 3.0$; $\gamma = 6.5$, 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.

3. Implementacija konvolucije

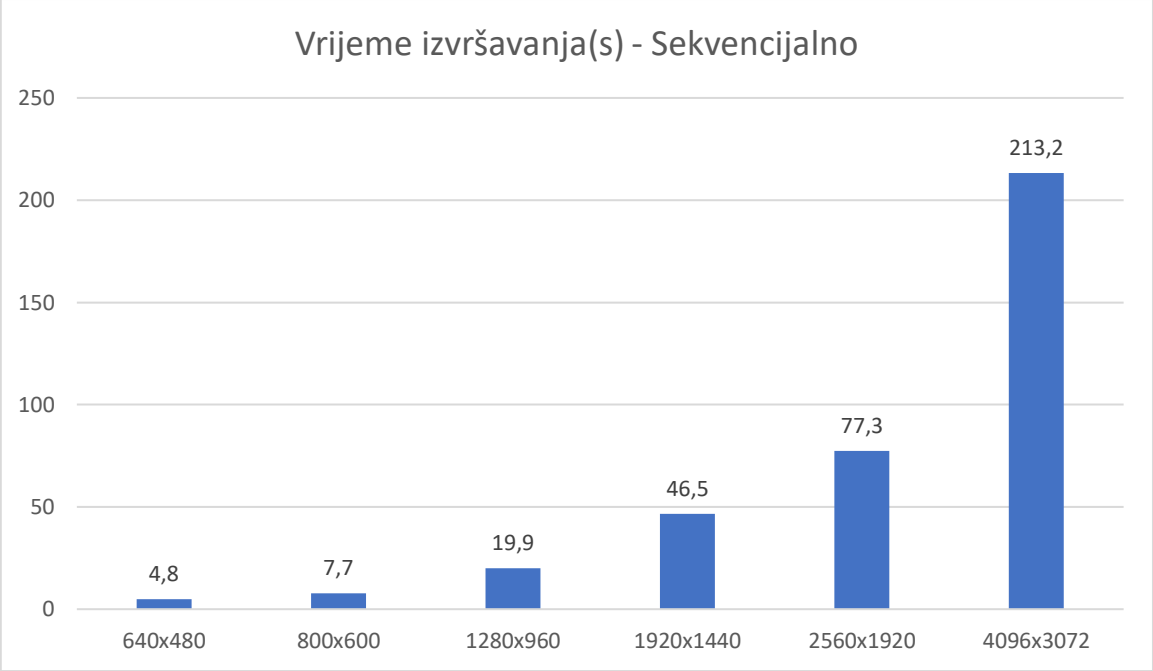
3.1. Kod

```
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);
}
```

3.2. Sekvencijalni rezultati

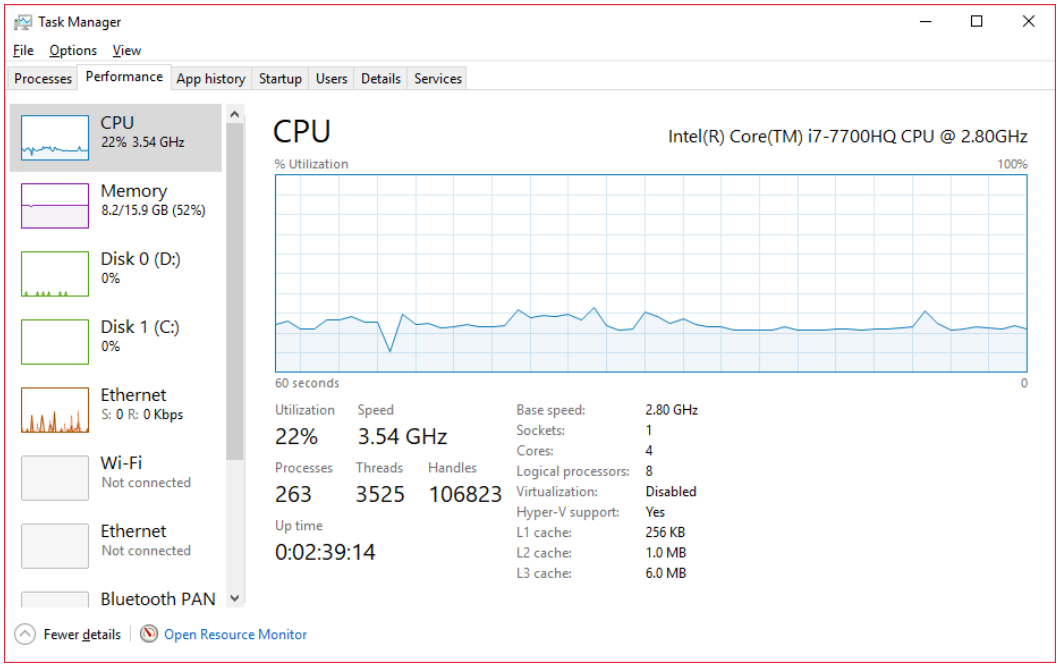
Funkcija prima tri parametra: matricu crno-bijele ulazne slike, matricu kernela i strukturu kojom je definirano hoće li funkcija biti izvedena sekvencijalno ili paralelno. U slučaju paralelnog izvođenja, istom strukturom se definira broj niti.

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

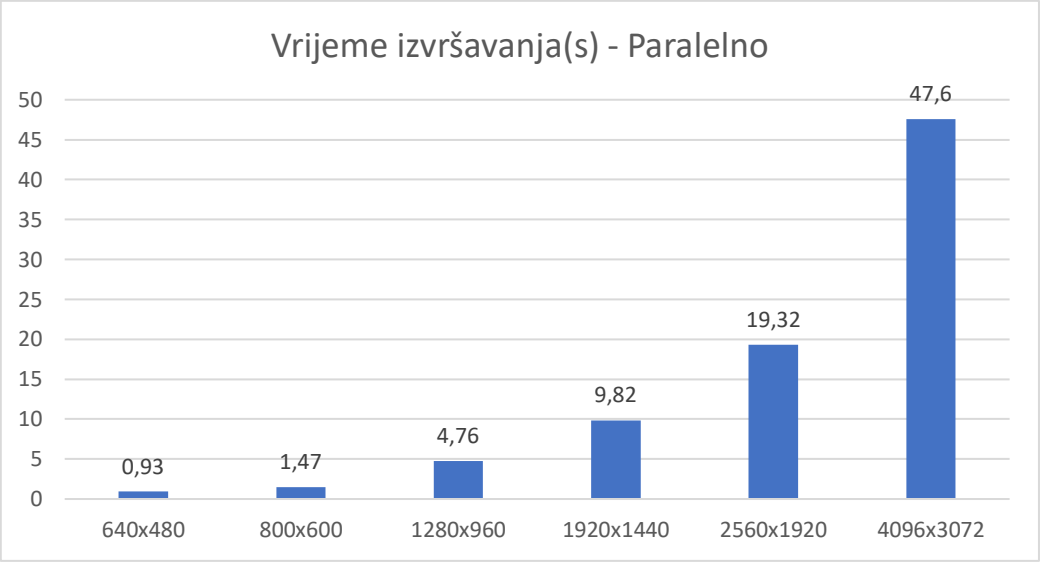
3.3. Kod paralelne podijele

```
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;
}
```

3.4. Paralelni rezultati

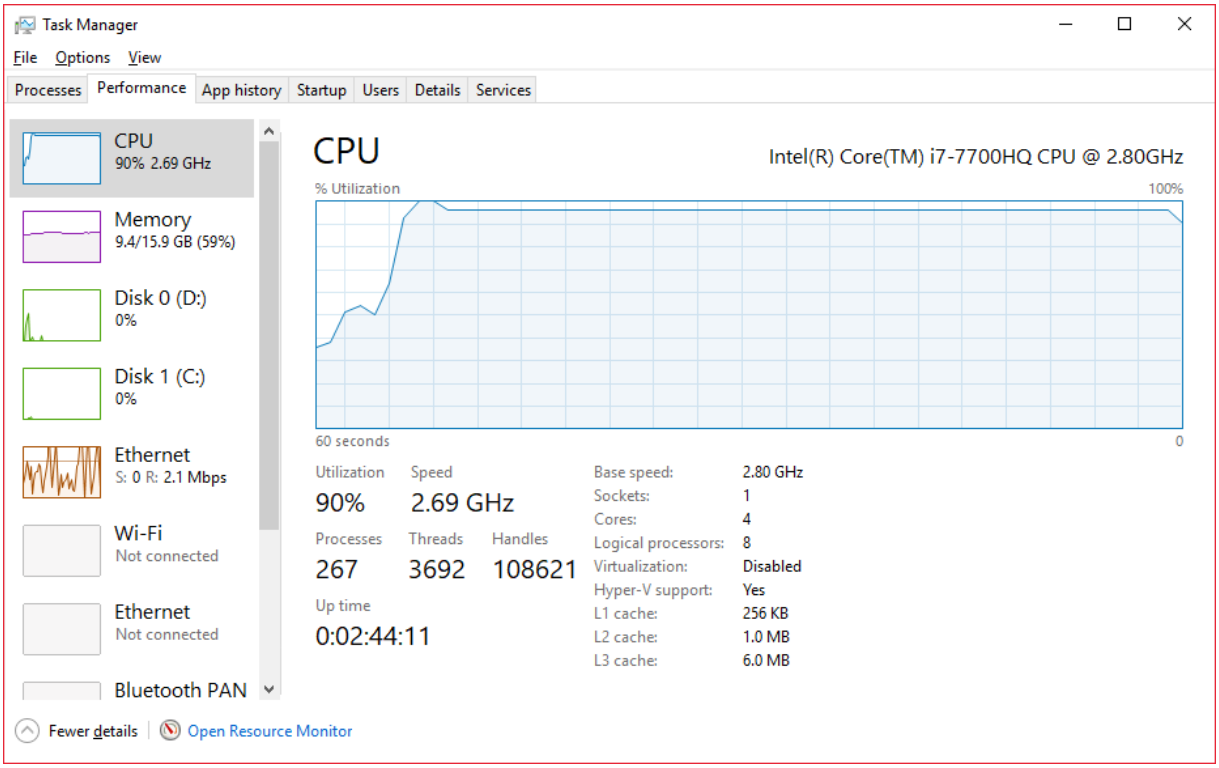
Za paralelno izvođenje programa se koristi ista funkcija za konvoluciju kao i kod sekvencijalne izvedbe uz promijenjene parametre koji definiraju postavke za paralelno izvođenje programa.

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.

4. CUDA

4.1. 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;
}
```

4.2. 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);
```

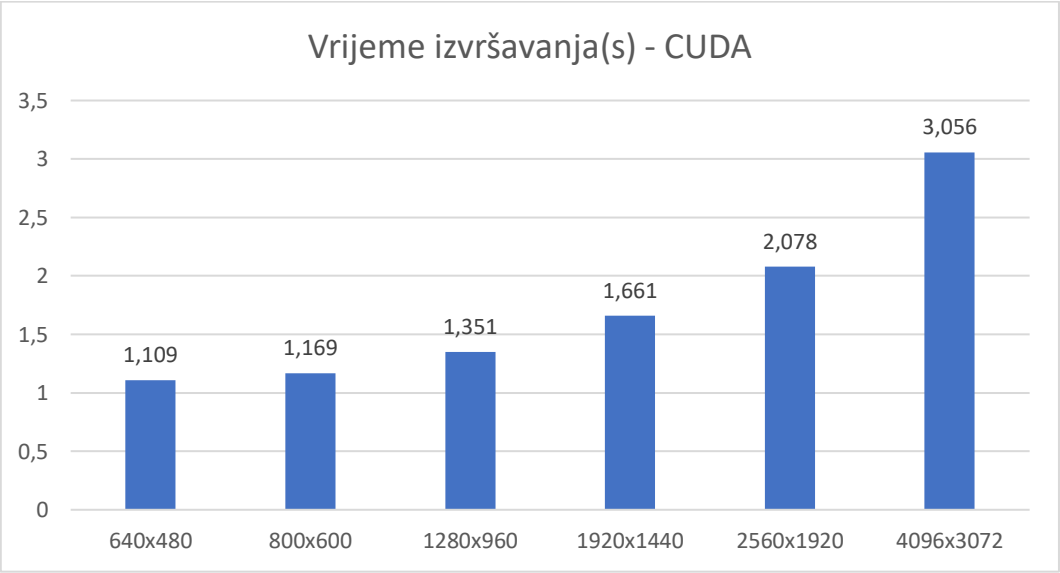
```
    output_d.release();

    return output;
}
```

4.3. 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

5. Prilog

5.1. 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
```

5.2. 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;
}
```

5.3. 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
```

5.4. 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                      3.14159265358979323846264338327950

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;
}
```

5.5. 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
```

5.6. 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;
}
```

```cpp
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;
}
```

5.7. inc/Image/Convolution.hpp

```cpp
#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
```

5.8. src/Image/Convolution.cpp

```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;
}
```

5.9. 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
```

5.10. 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
```

5.11. src/Utility/Stopwatch.cpp

```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;
}
```

5.12. 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;
}
```