SVEUČILIŠTE U SPLITU

FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

SEMINARSKI RAD

MEDIAN FILTAR REALIZIRAN U CUDI

Robert Blaškić

Split, lipanj 2021.

SADRŽAJ

1. UVOD

Grafička kartica (*eng. graphics processing unit, GPU*) jedan je od važnih dijelova računala čija je glavna zadaća paralelno izvođenje, a time i ubrzavanje programa. U ovom seminarskom radu opisana je prvenstveno obrada slike na GPU pomoću CUDA paralelne platforme i median filtra.

Median filtar najčešće se koristi za smanjenje šuma na crno-bijelim slikama. Algoritam se sastoji od određivanja kvadratnog prozora veličine 3x3 ili 5x5 ili sličnih te se svi pikseli u tom prozoru sortiraju po veličini. Zatim se iz sortiranog niza odredi medijan i upravo ta vrijednost predstavlja vrijednost piksela koji se nalazi u sredini prozora. Analogno se određuju i ostali pikseli slike. Što je veći prozor, bolje je, ali i sporije filtriranje. Median filtar dobro čuva korisne detalje na slici, ali nerijetko zamućuje oštre linije. Njegova najveća mana je u tome što je sporiji od ostalih filtara jer mu treba vremena za sortiranje svih susjednih članova kvadratnog prozora i to za svaki piksel slike.

Implementacija filtra napravit će se pomoću Nvidia CUDA Toolkit-a i OpenCV biblioteke. Nvidia CUDA Toolkit sadrži razne CUDA alate potrebne za paralelizaciju problema, a OpenCV biblioteka sadrži funkcije za učitavanje slike s procesora na grafičku karticu i obrnuto te za filtriranje same slike.

2. MEDIAN FILTAR

2.1. Vrste šumova

Slikovni šum je slučajna varijacija svjetline ili informacija o boji na slikama i obično je aspekt elektroničkog šuma. Može ga proizvesti senzor slike i sklop skenera ili digitalnog fotoaparata. Šum je nepoželjni nusprodukt snimanja slike koji zaklanja željene informacije. Može se kretati od gotovo neprimjetnih mrlja na digitalnoj fotografiji snimljenoj u dobrom svjetlu, do optičkih i radioastronomskih slika koje su gotovo u potpunosti mutne, a iz kojih se sofisticiranom obradom može dobiti mala količina informacija. Takva razina šuma bila bi neprihvatljiva na fotografiji jer bi bilo nemoguće odrediti i subjekt.

Vrste šumova su: Gaussov šum, *salt and pepper* šum, *shot* šum, šum kvantizacije (jednoliki šum), anizotropni šum te povremeni šum.

Glavni izvori Gaussovog šuma u digitalnim slikama nastaju tijekom snimanja. Senzoru je svojstven šum zbog razine osvjetljenja i vlastite temperature, a elektronički krugovi povezani na senzor ubrizgavaju vlastiti udio šuma elektroničkog kruga.

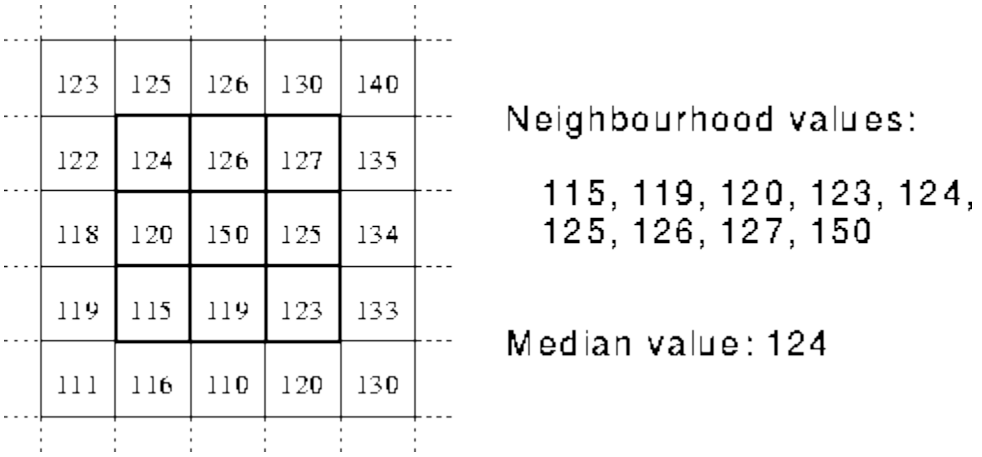
Raspodijeljeni ili impulzivni šum ponekad se naziva *salt and pepper* šum. Slika koja sadrži takav šum imat će tamne piksele u svijetlim područjima i svijetle piksele u tamnim regijama. Ovu vrstu šuma mogu uzrokovati pogreške analogno-digitalnog pretvarača, pogreške bitova u prijenosu itd. To se uglavnom može eliminirati oduzimanjem tamnog okvira, medijan filtriranjem, kombiniranim medijan filtriranjem i srednjih vrijednosti te interpolacijom oko tamnih ili svijetlih piksela. [1]

2.2. Median filtar

Median filtar obično se koristi za smanjenje šuma na slici. Sličan je mean filtru, međutim često radi bolji posao od prosječnog filtra za očuvanje korisnih detalja na slici.

Median filtar redom razmatra svaki piksel na slici i promatra susjedne vrijednosti kako bi odlučio je li reprezentativan za njegovu okolinu. Umjesto da vrijednost piksela jednostavno

zamijeni srednjom vrijednošću susjednih piksela, zamjenjuje je medijanom tih vrijednosti. Medijan se izračunava sortiranjem svih vrijednosti piksela iz okolnog susjedstva u numerički redoslijed, a zatim zamjenom piksela koji se razmatra srednjom vrijednosti piksela. Ako susjedstvo koje razmatra sadrži paran broj piksela, koristi se prosjek dviju srednjih vrijednosti piksela. Slika 3.1. ilustrira primjer izračuna.



Slika 2.1. Primjer izračuna median filtra

Kao što se može vidjeti sa slike, središnja vrijednost piksela 150 prilično je nereprezentativna za okolne piksele i zamijenjena je medijanom: 124. Ovdje se koristi kvadratura od 3 × 3, a veća susjedstva će proizvesti ozbiljnije zaglađivanje.

Izračunavanjem medijana vrijednosti susjedstva, median filtar ima dvije glavne prednosti u odnosu na mean filtar:

1) medijan je robusniji prosjek od srednje vrijednosti, pa jedan vrlo nereprezentativni piksel u susjedstvu neće značajno utjecati na vrijednost medijana;

2) budući da srednja vrijednost zapravo mora biti vrijednost jednog od piksela u susjedstvu, medijan filtar ne stvara nove nerealne vrijednosti piksela kad filtar prođe preko ruba. Iz tog razloga median filtar mnogo bolje čuva oštre rubove od mean filtra.

Općenito, median filtar omogućuje prolazak mnogo detalja visoke prostorne frekvencije, a istovremeno ostaje vrlo učinkovit u uklanjanju šuma na slikama gdje je postignuto manje od

polovice piksela u zaglađivanju. Kao posljedica toga, medijan filtriranje može biti manje učinkovito u uklanjanju šuma sa slika oštećenih Gaussovim šumom. Najčešće se median filtar koristi u obradi slika sa „*salt and pepper*“ šumom.

Jedan od glavnih problema s median filtrom je taj što je relativno skup i složen za izračunavanje. Da bi se pronašao medijan, potrebno je razvrstati sve vrijednosti u susjedstvu u numerički redoslijed, a to je relativno sporo, čak i kod brzih algoritama za sortiranje, poput *QuickSort*-a. Međutim, osnovni algoritam može se donekle poboljšati. [2]

3. NVIDIA CUDA TOOLKIT I OPENCV

3.1. CUDA

CUDA je paralelna računalna platforma i programsko sučelje koje omogućuje bolje iskorištenje grafičke kartice za opću namjenu. Programi se pišu u C/C++, Python-u, ali i drugim programskim jezicima. Paralelizam se očituje u korištenju ekstenzija i ključnih riječi karakterističnih za CUDA programe. [3]

Grafička kartica kompatibilna s CUDA platformom na kojoj se pokreće kod za median filtar je Nvidia GeForce GTX 850M. Njene specifikacije prikazane su na Slici 4.1.

```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.3\1_Utilities\deviceQuery\../../bin/win64/Debug/deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce GTX 850M"
  CUDA Driver Version / Runtime Version          11.3 / 11.3
  CUDA Capability Major/Minor version number:    5.0
  Total amount of global memory:                 2048 MBytes (2147483648 bytes)
  (005) Multiprocessors, (128) CUDA Cores/MP:    640 CUDA Cores
  GPU Max Clock rate:                            902 MHz (0.90 GHz)
  Memory Clock rate:                             1001 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 4 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.3, CUDA Runtime Version = 11.3, NumDevs = 1
Result = PASS
```

Slika 3.1. Grafička kartica NVIDIA GeForce GTX 850M

3.2. CUDA Toolkit

Nvidia Cuda Toolkit omogućuje razvojno okruženje za stvaranje ubrzanih GPU aplikacija visokih performansi. Njegovim korištenjem mogu se razviti, optimizirati i rasporediti aplikacije na ubrzanim GPU ugrađenim sustavima, stolnim računalima, platformama zasnovanim na oblaku i računalima visokih performansi (*eng. high performance computing, HPC*). Nvidia Cuda Toolkit sadrži upravljački program za CUDU, biblioteke za ubrzavanje GPU-a, alate za otklanjanje pogrešaka i optimizaciju CUDA aplikacija, kompajler za C i C++ programske jezike, biblioteku pomoću koje se kontrolira vrijeme izvršavanja pri izradi i implementaciji programskog koda na naprednijim arhitekturama računala, primjere CUDA programskog koda, itd.

Korištenjem ugrađenih alata za distribuciju izračuna na višestruke GPU konfiguracije mogu se razviti aplikacije prilagođene za jednu GPU radnu stanicu, ali i za radne stanice temeljene na oblaku s tisućama povezanih GPU-ova.

Moguće je izvršavati CUDA programe i kada računalo nema kompatibilnu Nvidia grafičku karticu, ali se kod vjerojatno neće ispravno pokrenuti. Naime Nvidia upravljački program sadrži biblioteke koje su potrebne za ispravan rad CUDA programa, a koje grafičke kartice drugih arhitektura nemaju. [4, 5]

3.3. OpenCV

OpenCV jedna je od najvećih biblioteka otvorenog izvora koja se koristi u području računalnog vida (*eng. computer vision*). Računalni vid je područje koje se bavi obradom, spremanjem i dohvatom određenih podataka iz slika. Ono je temelj umjetne inteligencije, a sveprisutno je i u robotici. Računalni vid povezuje i druga područja poput proširene stvarnosti (*eng. augmented reality*), dubokog učenja (*eng. deep learning*), praćenja objekata (*eng. object tracking*).

Glavni cilj OpenCV biblioteke je obrada slike u realnom vremenu, a neke od primjena navedene su u nastavku:

- video nadzor, primjerice detekcija pokreta na videu koja može identificirati provalnika;
- kreiranje digitalnog potpisa za PDF;
- proširena stvarnost, primjerice odabir određenih dijelova videa i njihova zamjena sadržajem po želji;
- duboko učenje, primjerice praćenje ljudi pomoću dronova;
- uređivanje slika, primjerice segmentacija pozadine u svrhu naglašavanja onog što je u prvom planu;
- detekcija te zamagljivanje lica;
- detekcija teksta na slikama;
- nadzor prometa;
- detekcija trake kod autonomnih vozila;
- interaktivna umjetnost;
- prepoznavanje objekata, primjerice anomalija u proizvodnom procesu (neobični oblici proizvoda),…

Tema ovog seminarskog rada je median filtar te se u tu svrhu koriste upravo funkcije iz OpenCV biblioteke. Postoje i druga razvojna okruženja za obradu slike kao što je TensorFlow, Keras, Caffe i Google Colab, ali OpenCV je najpopularnije. [6, 7]

4. PROGRAMSKI KOD I REZULTATI OBRADE FILTROM

4.1. Programski kod

Za realizaciju median filtra u CUDA okruženju korišten je sljedeći C++ programski kod:

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <opencv2/core.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/imgcodecs.hpp>
#include <opencv2/highgui.hpp>
#include <stdio.h>
#include <iostream>
#include <chrono>

using namespace cv;
using namespace std;
using namespace std::chrono;

#define WINDOW_SIZE 3
#define BLOCK 4
#define SIZE_OF_FILTER WINDOW_SIZE*WINDOW_SIZE

__host__ void  memoryInitCUDA(unsigned char* in_img_data, unsigned char* out_img_data, int imag
eHeight, int imageWidth, int Channels, long* time, bool isSharedMemory);
__host__ void medianFilterCPU(unsigned char* inputImageKernel, unsigned char* outputImageKernel
, int imageHeight, int imageWidth);
__global__ void medianFilterCUDA(unsigned char* inputImageKernel, unsigned char* outputImageKer
nel, int imageHeight, int imageWidth);
__global__ void medianFilterCUDAShared(unsigned char* inputImageKernel, unsigned char* outputIm
ageKernel, int imageHeight, int imageWidth);

int main()
{
    Mat img = imread("FESB1CV.bmp", IMREAD_GRAYSCALE);
    Mat resCPU = Mat::zeros(img.size(), CV_8UC1);
    Mat resGPU = Mat::zeros(img.size(), CV_8UC1);
    Mat resGPUShared = Mat::zeros(img.size(), CV_8UC1);

    long timeCPU, timeGPU, timeGPUShared;

    cout << "LOADED IMAGE INFO: " << endl;
```

```
    cout << "Image Height: " << img.rows << ", Image Width: " << img.cols << ", Image Channels:
 " << img.channels() << endl << endl;
    cout << "USED: " << WINDOW_SIZE << "x" << WINDOW_SIZE << " median filter" << endl << endl;
    cout << "MEASURED TIME:" << endl << endl;

    auto timeStart = high_resolution_clock::now();
    medianFilterCPU(img.data, resCPU.data, img.rows, img.cols);
    auto timeStop = high_resolution_clock::now();
    timeCPU = duration_cast<microseconds>(timeStop - timeStart).count();

    cout << "CPU time: " << timeCPU << " us = " << (float)timeCPU / 1000 << " ms" << endl;

    memoryInitCUDA(img.data, resGPU.data, img.rows, img.cols, img.channels(), &timeGPU, false);
    cout << "GPU time: " << timeGPU << " us = " << (float)timeGPU / 1000 << " ms" << endl;

    memoryInitCUDA(img.data, resGPUShared.data, img.rows, img.cols, img.channels(), &timeGPUSha
red, true);
    cout << "GPU - Shared Memory time: " << timeGPUShared << " us = " << (float)timeGPUShared /
 1000 << " ms" << endl << endl;

    imwrite("Filtered_Image_CPU.bmp", resCPU);
    imwrite("Filtered_Image_GPU.bmp", resGPU);
    imwrite("Filtered_Image_GPU_Shared.bmp", resGPUShared);

    cout << "Input image was successfully filtered by the Median filter!" << endl;
    system("pause");
    return 0;
}

__host__ void  memoryInitCUDA(unsigned char* in_img_data, unsigned char* out_img_data, int imag
eHeight, int imageWidth, int Channels, long* time, bool isSharedMemory)
{
    unsigned char* dev_in = NULL;
    unsigned char* dev_out = NULL;

    clock_t timeStart, timeEnd;

    dim3 dimBlock(BLOCK, BLOCK);
    dim3 dimGrid((int)ceil((float)imageWidth / (float)BLOCK),
        (int)ceil((float)imageHeight / (float)BLOCK));

    cudaMalloc((void**)&dev_in, (_int64)(imageHeight * imageWidth * Channels));
    cudaMalloc((void**)&dev_out, (_int64)(imageHeight * imageWidth * Channels));
```

```
    cudaMemcpy(dev_in, in_img_data, (_int64)(imageHeight * imageWidth * Channels), cudaMemcpyHo
stToDevice);
    cudaMemcpy(dev_out, out_img_data, (_int64)(imageHeight * imageWidth * Channels), cudaMemcpy
HostToDevice);

    auto timeStartInternal = high_resolution_clock::now();
    if (isSharedMemory == false)
        medianFilterCUDA << <dimGrid, dimBlock >> > (dev_in, dev_out, imageHeight, imageWidth);
    else
        medianFilterCUDAShared << <dimGrid, dimBlock >> > (dev_in, dev_out, imageHeight, imageW
idth);
    auto timeStopInternal = high_resolution_clock::now();

    *time = duration_cast<microseconds>(timeStopInternal - timeStartInternal).count();

    cudaMemcpy(in_img_data, dev_in, (_int64)(imageHeight * imageWidth * Channels), cudaMemcpyDe
viceToHost);
    cudaMemcpy(out_img_data, dev_out, (_int64)(imageHeight * imageWidth * Channels), cudaMemcpy
DeviceToHost);

    cudaFree(dev_in);
    cudaFree(dev_out);
}


__host__ void medianFilterCPU(unsigned char* inputImageKernel, unsigned char* outputImageKernel
, int imageHeight, int imageWidth)
{
    unsigned char filterVector[SIZE_OF_FILTER] = { 0 };
    for (int row = 0; row < imageHeight; row++) {
        for (int col = 0; col < imageWidth; col++) {
            if ((row == 0) || (col == 0) || (row == imageHeight - 1) || (col == imageWidth - 1)
)
                outputImageKernel[row * imageWidth + col] = 0;
            else {
                for (int x = 0; x < WINDOW_SIZE; x++) {
                    for (int y = 0; y < WINDOW_SIZE; y++) {
                        filterVector[x * WINDOW_SIZE + y] = inputImageKernel[(row + x - 1) * im
ageWidth + (col + y - 1)];
                    }
                }

                for (int i = 0; i < SIZE_OF_FILTER; i++) {
                    for (int j = i + 1; j < SIZE_OF_FILTER; j++)
                    {
```

```
                    if (filterVector[i] > filterVector[j]) {
                        char tmp = filterVector[i];
                        filterVector[i] = filterVector[j];
                        filterVector[j] = tmp;
                    }
                }
            }
            outputImageKernel[row * imageWidth + col] = filterVector[SIZE_OF_FILTER / 2];
        }
    }
}

__global__ void medianFilterCUDA(unsigned char* inputImageKernel, unsigned char* outputImageKer
nel, int imageHeight, int imageWidth)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned char filterVector[SIZE_OF_FILTER] = { 0 };
    if ((row == 0) || (col == 0) || (row == imageHeight - 1) || (col == imageWidth - 1))
        outputImageKernel[row * imageWidth + col] = 0;
    else {
        for (int x = 0; x < WINDOW_SIZE; x++) {
            for (int y = 0; y < WINDOW_SIZE; y++) {
                filterVector[x * WINDOW_SIZE + y] = inputImageKernel[(row + x - 1) * imageWidth
 + (col + y - 1)];
            }
        }
        for (int i = 0; i < SIZE_OF_FILTER; i++) {
            for (int j = i + 1; j < SIZE_OF_FILTER; j++) {
                if (filterVector[i] > filterVector[j]) {
                    char tmp = filterVector[i];
                    filterVector[i] = filterVector[j];
                    filterVector[j] = tmp;
                }
            }
        }
        outputImageKernel[row * imageWidth + col] = filterVector[SIZE_OF_FILTER / 2];
    }
}

__global__ void medianFilterCUDAShared(unsigned char* inputImageKernel, unsigned char* outputIm
ageKernel, int imageHeight, int imageWidth)
{
    int row = blockIdx.y * blockDim.y + threadIdx.y;
```

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

    __shared__ unsigned char sharedmem[(WINDOW_SIZE + 2)][(WINDOW_SIZE + 2)];

    bool is_x_left = (threadIdx.x == 0), is_x_right = (threadIdx.x == WINDOW_SIZE - 1);
    bool is_y_top = (threadIdx.y == 0), is_y_bottom = (threadIdx.y == WINDOW_SIZE - 1);

    if (is_x_left)
        sharedmem[threadIdx.x][threadIdx.y + 1] = 0;
    else if (is_x_right)
        sharedmem[threadIdx.x + 2][threadIdx.y + 1] = 0;
    if (is_y_top) {
        sharedmem[threadIdx.x + 1][threadIdx.y] = 0;
        if (is_x_left)
            sharedmem[threadIdx.x][threadIdx.y] = 0;
        else if (is_x_right)
            sharedmem[threadIdx.x + 2][threadIdx.y] = 0;
    }
    else if (is_y_bottom) {
        sharedmem[threadIdx.x + 1][threadIdx.y + 2] = 0;
        if (is_x_right)
            sharedmem[threadIdx.x + 2][threadIdx.y + 2] = 0;
        else if (is_x_left)
            sharedmem[threadIdx.x][threadIdx.y + 2] = 0;
    }

    sharedmem[threadIdx.x + 1][threadIdx.y + 1] = inputImageKernel[row * imageWidth + col];

    if (is_x_left && (col > 0))
        sharedmem[threadIdx.x][threadIdx.y + 1] = inputImageKernel[row * imageWidth + (col - 1)
];
    else if (is_x_right && (col < imageWidth - 1))
        sharedmem[threadIdx.x + 2][threadIdx.y + 1] = inputImageKernel[row * imageWidth + (col
+ 1)];
    if (is_y_top && (row > 0)) {
        sharedmem[threadIdx.x + 1][threadIdx.y] = inputImageKernel[(row - 1) * imageWidth + col
];
        if (is_x_left)
            sharedmem[threadIdx.x][threadIdx.y] = inputImageKernel[(row - 1) * imageWidth + (co
l - 1)];
        else if (is_x_right)
            sharedmem[threadIdx.x + 2][threadIdx.y] = inputImageKernel[(row - 1) * imageWidth +
 (col + 1)];
    }
    else if (is_y_bottom && (row < imageHeight - 1)) {
```

```
        sharedmem[threadIdx.x + 1][threadIdx.y + 2] = inputImageKernel[(row + 1) * imageWidth +
 col];
        if (is_x_right)
            sharedmem[threadIdx.x + 2][threadIdx.y + 2] = inputImageKernel[(row + 1) * imageWid
th + (col + 1)];
        else if (is_x_left)
            sharedmem[threadIdx.x][threadIdx.y + 2] = inputImageKernel[(row + 1) * imageWidth +
 (col - 1)];
    }

    __syncthreads();


    if (SIZE_OF_FILTER == 9)
    {
        unsigned char filterVector[9] = { sharedmem[threadIdx.x][threadIdx.y], sharedmem[thread
Idx.x + 1][threadIdx.y], sharedmem[threadIdx.x + 2][threadIdx.y],
                    sharedmem[threadIdx.x][threadIdx.y + 1], sharedmem[threadIdx.x + 1][threadId
x.y + 1], sharedmem[threadIdx.x + 2][threadIdx.y + 1],
                    sharedmem[threadIdx.x][threadIdx.y + 2], sharedmem[threadIdx.x + 1][threadId
x.y + 2], sharedmem[threadIdx.x + 2][threadIdx.y + 2] };


        {
            for (int i = 0; i < SIZE_OF_FILTER; i++) {
                for (int j = i + 1; j < SIZE_OF_FILTER; j++)
                {

                    if (filterVector[i] > filterVector[j]) {
                        char tmp = filterVector[i];
                        filterVector[i] = filterVector[j];
                        filterVector[j] = tmp;
                    }
                }
            }
            outputImageKernel[row * imageWidth + col] = filterVector[SIZE_OF_FILTER / 2];
        }

    }
    else if (SIZE_OF_FILTER == 16)
    {
        unsigned char filterVector[16] = { sharedmem[threadIdx.x][threadIdx.y], sharedmem[threa
dIdx.x + 1][threadIdx.y], sharedmem[threadIdx.x + 2][threadIdx.y],sharedmem[threadIdx.x + 3][th
readIdx.y],
```

```
                    sharedmem[threadIdx.x][threadIdx.y + 1], sharedmem[threadIdx.x + 1][threadId
x.y + 1], sharedmem[threadIdx.x + 2][threadIdx.y + 1],sharedmem[threadIdx.x + 3][threadIdx.y +
1],
                    sharedmem[threadIdx.x][threadIdx.y + 2], sharedmem[threadIdx.x + 1][threadId
x.y + 2], sharedmem[threadIdx.x + 2][threadIdx.y + 2],sharedmem[threadIdx.x + 3][threadIdx.y +
2],
                     sharedmem[threadIdx.x][threadIdx.y + 3], sharedmem[threadIdx.x + 1][threadI
dx.y + 3], sharedmem[threadIdx.x + 2][threadIdx.y + 3],sharedmem[threadIdx.x + 3][threadIdx.y +
 3] };


        {
            for (int i = 0; i < SIZE_OF_FILTER; i++) {
                for (int j = i + 1; j < SIZE_OF_FILTER; j++)
                {

                    if (filterVector[i] > filterVector[j]) {
                        char tmp = filterVector[i];
                        filterVector[i] = filterVector[j];
                        filterVector[j] = tmp;
                    }
                }
            }
            outputImageKernel[row * imageWidth + col] = filterVector[SIZE_OF_FILTER / 2];
        }
    }
    else if (SIZE_OF_FILTER == 25)
    {
        unsigned char filterVector[25] = { sharedmem[threadIdx.x][threadIdx.y], sharedmem[threa
dIdx.x + 1][threadIdx.y], sharedmem[threadIdx.x + 2][threadIdx.y],sharedmem[threadIdx.x + 3][th
readIdx.y],sharedmem[threadIdx.x + 4][threadIdx.y],
                    sharedmem[threadIdx.x][threadIdx.y + 1], sharedmem[threadIdx.x + 1][threadId
x.y + 1], sharedmem[threadIdx.x + 2][threadIdx.y + 1],sharedmem[threadIdx.x + 3][threadIdx.y +
1],sharedmem[threadIdx.x + 4][threadIdx.y + 1],
                    sharedmem[threadIdx.x][threadIdx.y + 2], sharedmem[threadIdx.x + 1][threadId
x.y + 2], sharedmem[threadIdx.x + 2][threadIdx.y + 2],sharedmem[threadIdx.x + 3][threadIdx.y +
2],sharedmem[threadIdx.x + 4][threadIdx.y + 2],
                     sharedmem[threadIdx.x][threadIdx.y + 3], sharedmem[threadIdx.x + 1][threadI
dx.y + 3], sharedmem[threadIdx.x + 2][threadIdx.y + 3],sharedmem[threadIdx.x + 3][threadIdx.y +
 3], sharedmem[threadIdx.x + 4][threadIdx.y + 3],
                     sharedmem[threadIdx.x][threadIdx.y + 4], sharedmem[threadIdx.x + 1][threadI
dx.y + 4], sharedmem[threadIdx.x + 2][threadIdx.y + 4],sharedmem[threadIdx.x + 3][threadIdx.y +
 4], sharedmem[threadIdx.x + 4][threadIdx.y + 4] };
```

```
        {
            for (int i = 0; i < SIZE_OF_FILTER; i++) {
                for (int j = i + 1; j < SIZE_OF_FILTER; j++)
                {

                    if (filterVector[i] > filterVector[j]) {
                        char tmp = filterVector[i];
                        filterVector[i] = filterVector[j];
                        filterVector[j] = tmp;
                    }
                }
            }
            outputImageKernel[row * imageWidth + col] = filterVector[SIZE_OF_FILTER / 2];
        }
    }
    else if (SIZE_OF_FILTER == 49)
    {
        unsigned char filterVector[49] = { sharedmem[threadIdx.x][threadIdx.y], sharedmem[threa
dIdx.x + 1][threadIdx.y], sharedmem[threadIdx.x + 2][threadIdx.y],sharedmem[threadIdx.x + 3][th
readIdx.y],sharedmem[threadIdx.x + 4][threadIdx.y],sharedmem[threadIdx.x + 5][threadIdx.y],shar
edmem[threadIdx.x + 6][threadIdx.y],
                    sharedmem[threadIdx.x][threadIdx.y + 1], sharedmem[threadIdx.x + 1][threadId
x.y + 1], sharedmem[threadIdx.x + 2][threadIdx.y + 1],sharedmem[threadIdx.x + 3][threadIdx.y +
1],sharedmem[threadIdx.x + 4][threadIdx.y + 1],sharedmem[threadIdx.x + 5][threadIdx.y+1],shared
mem[threadIdx.x + 6][threadIdx.y+1],
                    sharedmem[threadIdx.x][threadIdx.y + 2], sharedmem[threadIdx.x + 1][threadId
x.y + 2], sharedmem[threadIdx.x + 2][threadIdx.y + 2],sharedmem[threadIdx.x + 3][threadIdx.y +
2],sharedmem[threadIdx.x + 4][threadIdx.y + 2],sharedmem[threadIdx.x + 5][threadIdx.y+2],shared
mem[threadIdx.x + 6][threadIdx.y+2],
                     sharedmem[threadIdx.x][threadIdx.y + 3], sharedmem[threadIdx.x + 1][threadI
dx.y + 3], sharedmem[threadIdx.x + 2][threadIdx.y + 3],sharedmem[threadIdx.x + 3][threadIdx.y +
 3], sharedmem[threadIdx.x + 4][threadIdx.y + 3],sharedmem[threadIdx.x + 5][threadIdx.y+3],shar
edmem[threadIdx.x + 6][threadIdx.y+3],
                     sharedmem[threadIdx.x][threadIdx.y + 4], sharedmem[threadIdx.x + 1][threadI
dx.y + 4], sharedmem[threadIdx.x + 2][threadIdx.y + 4],sharedmem[threadIdx.x + 3][threadIdx.y +
 4], sharedmem[threadIdx.x + 4][threadIdx.y + 4], sharedmem[threadIdx.x + 5][threadIdx.y+4],sha
redmem[threadIdx.x + 6][threadIdx.y+4],
                     sharedmem[threadIdx.x][threadIdx.y + 5], sharedmem[threadIdx.x + 1][threadI
dx.y + 5], sharedmem[threadIdx.x + 2][threadIdx.y + 5],sharedmem[threadIdx.x + 3][threadIdx.y +
 5], sharedmem[threadIdx.x + 4][threadIdx.y + 5], sharedmem[threadIdx.x + 5][threadIdx.y + 5],s
haredmem[threadIdx.x + 6][threadIdx.y + 5],
                     sharedmem[threadIdx.x][threadIdx.y + 6], sharedmem[threadIdx.x + 1][threadI
dx.y + 6], sharedmem[threadIdx.x + 2][threadIdx.y + 6],sharedmem[threadIdx.x + 3][threadIdx.y +
 6], sharedmem[threadIdx.x + 4][threadIdx.y + 6], sharedmem[threadIdx.x + 5][threadIdx.y + 6],s
haredmem[threadIdx.x + 6][threadIdx.y + 6], };
```

```
        {
            for (int i = 0; i < SIZE_OF_FILTER; i++) {
                for (int j = i + 1; j < SIZE_OF_FILTER; j++)
                {

                    if (filterVector[i] > filterVector[j]) {
                        char tmp = filterVector[i];
                        filterVector[i] = filterVector[j];
                        filterVector[j] = tmp;
                    }
                }
            }
            outputImageKernel[row * imageWidth + col] = filterVector[SIZE_OF_FILTER / 2];
        }
    }
    else if (SIZE_OF_FILTER == 81)
    {
        unsigned char filterVector[81] = { sharedmem[threadIdx.x][threadIdx.y], sharedmem[threa
dIdx.x + 1][threadIdx.y], sharedmem[threadIdx.x + 2][threadIdx.y],sharedmem[threadIdx.x + 3][th
readIdx.y],sharedmem[threadIdx.x + 4][threadIdx.y],sharedmem[threadIdx.x + 5][threadIdx.y],shar
edmem[threadIdx.x + 6][threadIdx.y],sharedmem[threadIdx.x + 7][threadIdx.y],sharedmem[threadIdx
.x + 8][threadIdx.y],
                  sharedmem[threadIdx.x][threadIdx.y + 1], sharedmem[threadIdx.x + 1][threadId
x.y + 1], sharedmem[threadIdx.x + 2][threadIdx.y + 1],sharedmem[threadIdx.x + 3][threadIdx.y +
1],sharedmem[threadIdx.x + 4][threadIdx.y + 1],sharedmem[threadIdx.x + 5][threadIdx.y + 1],shar
edmem[threadIdx.x + 6][threadIdx.y + 1],sharedmem[threadIdx.x + 7][threadIdx.y+1],sharedmem[thr
eadIdx.x + 8][threadIdx.y+1],
                  sharedmem[threadIdx.x][threadIdx.y + 2], sharedmem[threadIdx.x + 1][threadId
x.y + 2], sharedmem[threadIdx.x + 2][threadIdx.y + 2],sharedmem[threadIdx.x + 3][threadIdx.y +
2],sharedmem[threadIdx.x + 4][threadIdx.y + 2],sharedmem[threadIdx.x + 5][threadIdx.y + 2],shar
edmem[threadIdx.x + 6][threadIdx.y + 2],sharedmem[threadIdx.x + 7][threadIdx.y + 2],sharedmem[t
hreadIdx.x + 8][threadIdx.y + 2],
                   sharedmem[threadIdx.x][threadIdx.y + 3], sharedmem[threadIdx.x + 1][threadI
dx.y + 3], sharedmem[threadIdx.x + 2][threadIdx.y + 3],sharedmem[threadIdx.x + 3][threadIdx.y +
 3], sharedmem[threadIdx.x + 4][threadIdx.y + 3],sharedmem[threadIdx.x + 5][threadIdx.y + 3],sh
aredmem[threadIdx.x + 6][threadIdx.y + 3],sharedmem[threadIdx.x + 7][threadIdx.y + 3],sharedmem
[threadIdx.x + 8][threadIdx.y + 3],
                   sharedmem[threadIdx.x][threadIdx.y + 4], sharedmem[threadIdx.x + 1][threadI
dx.y + 4], sharedmem[threadIdx.x + 2][threadIdx.y + 4],sharedmem[threadIdx.x + 3][threadIdx.y +
 4], sharedmem[threadIdx.x + 4][threadIdx.y + 4], sharedmem[threadIdx.x + 5][threadIdx.y + 4],s
haredmem[threadIdx.x + 6][threadIdx.y + 4],sharedmem[threadIdx.x + 7][threadIdx.y + 4],sharedme
m[threadIdx.x + 8][threadIdx.y + 4],
```

```
                    sharedmem[threadIdx.x][threadIdx.y + 5], sharedmem[threadIdx.x + 1][threadI
dx.y + 5], sharedmem[threadIdx.x + 2][threadIdx.y + 5],sharedmem[threadIdx.x + 3][threadIdx.y +
 5], sharedmem[threadIdx.x + 4][threadIdx.y + 5], sharedmem[threadIdx.x + 5][threadIdx.y + 5],s
haredmem[threadIdx.x + 6][threadIdx.y + 5],sharedmem[threadIdx.x + 7][threadIdx.y + 5],sharedme
m[threadIdx.x + 8][threadIdx.y + 5],
                    sharedmem[threadIdx.x][threadIdx.y + 6], sharedmem[threadIdx.x + 1][threadI
dx.y + 6], sharedmem[threadIdx.x + 2][threadIdx.y + 6],sharedmem[threadIdx.x + 3][threadIdx.y +
 6], sharedmem[threadIdx.x + 4][threadIdx.y + 6], sharedmem[threadIdx.x + 5][threadIdx.y + 6],s
haredmem[threadIdx.x + 6][threadIdx.y + 6],sharedmem[threadIdx.x + 7][threadIdx.y + 6],sharedme
m[threadIdx.x + 8][threadIdx.y + 6],
                    sharedmem[threadIdx.x][threadIdx.y + 7], sharedmem[threadIdx.x + 1][threadI
dx.y + 7], sharedmem[threadIdx.x + 2][threadIdx.y + 7],sharedmem[threadIdx.x + 3][threadIdx.y +
 7], sharedmem[threadIdx.x + 4][threadIdx.y + 7], sharedmem[threadIdx.x + 5][threadIdx.y + 7],s
haredmem[threadIdx.x + 6][threadIdx.y + 7],sharedmem[threadIdx.x + 7][threadIdx.y + 7],sharedme
m[threadIdx.x + 8][threadIdx.y + 7],
                    sharedmem[threadIdx.x][threadIdx.y + 8], sharedmem[threadIdx.x + 1][threadI
dx.y + 8], sharedmem[threadIdx.x + 2][threadIdx.y + 8],sharedmem[threadIdx.x + 3][threadIdx.y +
 8], sharedmem[threadIdx.x + 4][threadIdx.y + 8], sharedmem[threadIdx.x + 5][threadIdx.y + 8],s
haredmem[threadIdx.x + 6][threadIdx.y + 8],sharedmem[threadIdx.x + 7][threadIdx.y + 8],sharedme
m[threadIdx.x + 8][threadIdx.y + 8] };


        {
            for (int i = 0; i < SIZE_OF_FILTER; i++) {
                for (int j = i + 1; j < SIZE_OF_FILTER; j++)
                {

                    if (filterVector[i] > filterVector[j]) {
                        char tmp = filterVector[i];
                        filterVector[i] = filterVector[j];
                        filterVector[j] = tmp;
                    }
                }
            }
            outputImageKernel[row * imageWidth + col] = filterVector[SIZE_OF_FILTER / 2];
        }
    }

    else;


}
```

4.2. Objašnjenje programskog koda

main( ) funkcija: Učitavanje ulazne slike obavlja se naredbom *imread* s modom učitavanja jednokanalnog *grayscale*-a. Obavlja se deklaracija i inicijalizacija rezultantnih matrica slika nadopunom nulama konstantom *CV_8UC1* za jednokanalne slike. Nakon učitavanja ulazne slike obavlja se ispis parametara slike. Zatim se obavlja pozivanje *host* funkcije *medianFilterCPU* koja se izvodi na CPU i pripadajućim resursima (memorijom). Mjerenje vremena trajanja izvršavanja algoritma na različitim arhitekturama CPU/GPU/GPU s dijeljenom memorijom obavlja se s *high_resolution_clock* jer se može mjeriti sa skalom reda veličine mikrosekundi (µs). Za poziv median filtra na GPU arhitekturu potrebno je pripremiti podatke za obradu GPU (obaviti prebacivanje iz *host* memorije u *device* memoriju), što se obavlja pozivanjem *host* funkcije *memoryInitCUDA*. Njom se prosljeđuju pokazivači pa se time i rezultati obrade (rezultat obrade se u istoj toj funkciji prebacuje iz *device* memorije u *host* memoriju). Upisivanje rezultantnih podataka dobivenih tim filtrom u rezultirajuću sliku obavlja se funkcijom *imwrite*. Koriste se definicije *WINDOW_SIZE* za specificiranje 1D veličine korištenog kvadratičnog filtra, *BLOCK* za specificiranje 1D veličine blokova za definiranje ukupnog broja blokova u *grid*-ovima prilikom poziva *kernel-a* median filtra za izvršavanje na GPU *hardware*-u, te *SIZE_OF_FILTER* kao 2D veličina lineariziranog kvadratičnog filtra.

medianFilterCPU( ) host funkcija: Kako se radi s tipom podataka *unsigned char* (odnosno njihovim nizovima) koristi se filtar koji je lineariziran vektorskom strukturom istog tipa *unsigned char* veličine *SIZE_OF_FILTER* inicijaliziran s 0. Koriste se uređeni parovi (*row*, *col*) kao indeksi dvostruke for petlje za prelazak kroz svaki piksel učitane slike preko *unsigned char* niza. *If* blokom se provjerava nalaze li se indeksi *row* i *col* na rubu slike. Ukoliko je uvjet istinit, taj rub slike se nadopunjuje 0, odnosno obavlja se *zero padding*, na način da se stvori okvir oko slike popunjen isključivo nulama (kako bi se osigurao pouzdaniji rad filtra). Za izračun medijana na rubnim pikselima slike uzimaju se u obzir susjedni pikseli, pa su tim uvjetom osigurani i takvi pikseli kako se ne bi pristupalo nedozvoljenim dijelovima memorije i s djelomičnim prozorom filtra. Ukoliko uvjet nije istinit, indeksi se ne nalaze na rubu, nego unutar slike, pa se inicijalizira još jedna dvostruka petlja kojom se osigurava punjenje prozora filtra pikselima učitane slike (praktički se slika skenira) te se nakon svakog skeniranja taj prozor pomiče za *col*+1, tj. u desno sve dok ne dođe

do kraja reda piksela slike, što je određeno širinom slike *imageWidth*. Nakon prolaska te širine prozor se vraća na početak novog sljedećeg retka učitane slike i postupak se ponavlja. Učitavanjem piksela sa slike u prozor filtra, za ostvarivanje medijan operacije nužno je prvo poredati piksele prema njihovim vrijednostima ili od rastućeg ili od padajućeg niza. Navedeno sortiranje obavlja se *BubbleSort* algoritmom složenosti $O(n^2)$ tako što se svakom iteracijom redaju elementi po redu sve dok se ne poredaju u monotonom stanju (rast/pad vrijednosti bez prekida). Temelj tog algoritma je očito *Swap* funkcija dvaju elemenata tog niza filtra. Medijan je zapravo element srednje pozicije poredanog niza, te se tim elementom prebrisuje odgovarajući piksel ulazne slike te se time generira izlazna slika. Kako se radi o nizu, preko pokazivača se vraća izračunati niz izlazne slike *outputImageKernel*.

memoryInitCUDA( ) host funkcija: Deklaracija i inicijalizacija niza *dev_in* i *dev_out* koji služe kao nizovi za prijenos podataka s *host* memorije u *device* memoriju. Osim tog definiraju se trodimenzionalne veličine *block*-a (*BLOCK***BLOCK**1) i *grid*-a (*imageWidth*/*BLOCK* * *imageHeight*/*BLOCK* * 1). Za alociranje memorijskog prostora na GPU koristi se *cudaMalloc* čija veličina ovisi o visini, širini slike i broju kanala slike. Kako bi GPU mogao pristupati podacima za obradu učitane slike, ti se podaci moraju prebaciti s *host* memorije u *device* memoriju koristeći funkciju *cudaMemcpy* u modu *cudaMemcpyHostToDevice*. Može se primijetiti da su ulazni parametar te matične funkcije podaci ulazne slike koji se kopiraju na prethodno deklarirani *dev_in*. Ovisno o tome je li pri pozivu matične *host* funkcije specifirano *boolean* tipom radi li sa *shared* memorijom ili bez nje, poziva se *kernel* median filtra koji će se izvršavati na GPU arhitekturi. *Kernel* se poziva operatorom <<< >>> u kojim se specificiraju broj *grid*-ova i broj blokova koji će se koristiti prilikom rada filtra. Nakon obrade median filtrom, dobiveni podaci se s *device* memorije prebacuju u *host* memoriju koristeći isto *cudaMemcpy* suprotnim modom *cudaMemcpyDeviceToHost*. Za dealociranje memorije na GPU memoriji koristi se *cudaFree*.

medianFilterCUDA( ) global funkcija: Uređeni par indeksa (*col*, *row*) dobiva se izračun indeksa određenog *thread*-a koji trenutno radi s istima. Uočimo da se radi o 2D adresiranju *thread*-a. GPU prolazi sliku paralelnim učitavanjem *row* i *col* piksela ulazne slike u različitim *thread*-ovima. Daljnji algoritam analogan je algoritmu napisanom za obradu median filtrom na CPU arhitekturi.

medianFilterCUDAShared( ) global funkcija: Uređeni par indeksa (*col*, *row*) dobiva se izračun indeksa određenog *thread*-a koji trenutno radi s istima. Uočimo da se radi o 2D adresiranju *thread*-a. GPU prolazi sliku paralelnim učitavanjem *row* i *col* piksela ulazne slike u različitim *thread*-ovima. U ovoj funkciji median filtar se realizira posredstvom dijeljene memorije kako bi više *thread*-ova sinkronizacijom unutar istog bloka moglo pristupiti zajedničkim podacima. To je dosta korisno ukoliko je lokalna memorija kojom se osigurava rad s pripadajućim kontekstima pojedinih *thread*-ova preopterećena. Deklaracija memorijskog prostora u dijeljenoj memoriji obavljena je matricom *sharedmem* veličine *WINDOW_SIZE*+2 (širina i visina). Dodaje se 2 jer se i horizontalno i vertikalno doda susjedni rub izvan prozora koji će biti popunjen nulama. Definiraju se *boolean* vrijednosti za specifikaciju rubova u 2D matrici piksela (x, y). Serijom *if-else* grananja obavlja se *zero padding* okvira oko samog prozora filtra koji se sprema u alocirani prostor na dijeljenoj memoriji *block*-a *sharedmem*. Ukoliko je to izvršeno, unutar memorijskog prostora za sam prozor filtra upisuju se pikseli skenirani s ulazne slike te se pozivom serije *if-else* grananja popunjavaju i rubni dijelovi prozora u dijeljenoj memoriji. Vrlo je bitan korak (budući da tu memoriju koristi više dretvi istog bloka) sinkronizacija svih tih dretvi kad naiđu na definiranu barijeru, a suprotno može doći do nedozvoljenih pristupa toj zajedničkoj dijeljenoj memoriji te time do stvaranja nekonzistentnosti podataka u dijeljenoj memoriji kojoj pristupaju sve dretve za daljnju obradu međupodataka. Sinkronizacija dretvi obavlja se CUDA naredbom *__syncthreads*( ). Nadalje, ovisno o veličini median filtra, definiramo vektor prozora filtra u kojem se pohranjuju podaci iz te dijeljenje memorije. Ovdje se ujedno vidi bitan značaj sinkronizacije dretvi i same konzistentnosti podataka. Daljnji algoritam je analogan onom za CPU implementaciju dobivanja i pohrane medijana u piksel izlazne slike, koji se temelji na dvije osnovne operacije: *BubbleSort* sortiranje niza i izračun medijana tog niza i njegovu pohranu na odgovarajući piksel izlazne slike.

4.3. Rezultati obrade median filtrom

Implementirani algoritam za median filtar za arhitekture CPU (sekvencijalnog izvođenja) i GPU (paralelnog izvođenja, s ili bez dijeljene memorije) optimalan je za *bitmap* jednokanalne (crno-bijele) slike proizvoljne rezolucije. Implementirani su median filtri dimenzija 3x3, 4x4, 5x5, 7x7 i 9x9. Složenost algoritma median filtra zapravo ponajviše ovisi o algoritmu za sortiranje. U

ovom kodu korišten je *BubbleSort* složenosti $O(n^2)$. Prema tome, operacija median filtra može se smatrati nelinearnom operacijom kojom se smanjuje šum slike (*eng. salt-pepper noise*), tj. povećava signal-šum omjer (*eng. Signal-Noise Ratio, SNR*), a pritom su slike sklone zamagljivanju (*eng. blur*), tj. izobličavanju ravnih linija i bridova objekata na slici.

Kao ulazna slika korištena je slika FESB zgrade s naknadno dodanim šumom kojim se demonstrativno želi pokazati moć ovakvog filtra. Navedena slika je pretvorena u *bitmap* (.bmp) format i prebačena s RGB u crno-bijeli format sustava boja (iz trokanalne slike stvorena je jednokanalna slika). Rezolucija slike je 487x730. Korišten je median filtar dimenzije 3x3 za implementaciju CPU/GPU/GPU sa dijeljenom memorijom. Za GPU implementaciju nužno je definirani veličinu bloka kojom se specificiraju trodimenzionalne veličine *grid*-a i bloka prilikom pozivanja *kernela* za izvođenje operacije median filtra nad učitanom slikom.

Slika 4.1. FESB zgrada, rezolucija 487x730, jedno-kanalna slika

Primjerice, za filtar veličine 3x3, šumovi na slici bit će efikasno uklonjeni, ali će se ujedno uništiti linearne strukture debljine 1 piksel, dok se pri veličini 7x7 uništavaju linearne strukture slike debljine do čak i 3 piksela. Razlog uništavanja linearnih struktura je taj što se za obradu medijan filtrom uz odgovarajući piksel slike uzimaju u obzir i susjedni pikseli slike, te se na osnovu

njihovog medijana generira vrijednost. Time se omogućuje uklanjanje impulsnih smetnji na slici (maksimalne i minimalne vrijednosti zasićenja piksela) tako što se „prebace“ u vrijednosti okolnih piksela obuhvaćenih filtrom, ali time se ujedno utječe i na vrlo tanke linearne strukture slike, tako što ti pikseli linije budu potisnuti zbog ostalih susjednih piksela. To je tzv. *blur* efekt, tj. izobličenje linearnosti pojedinih objekata u slici.

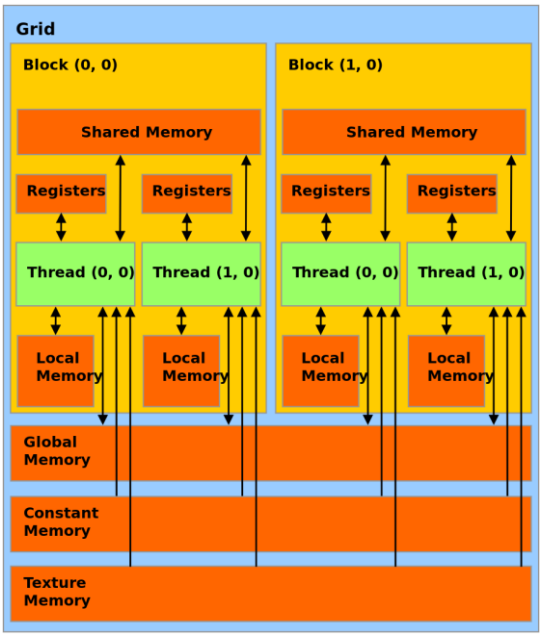
Rezultati obrade koristeći median filtar različitih dimenzija:

3x3	5x5
	
7x7	9x9
	

5. KRITIČKA ANALIZA PERFORMANSI

5.1. Analiza GPU arhitekture

Slika je zapravo dvodimenzionalna matrica čiji je osnovni element piksel (engleska složenica od riječi slika i element). Dakle, piksel, kao što je očekivano, predstavlja najmanju jedinicu (element) slike. Kako bi se grafičkoj kartici omogućio pristup pojedinim pikselima ulazne slike za obradu i njenu pohranu u pojedinim dretvama grafičke kartice, potrebno je obaviti odgovarajuću strukturalnu organizaciju resursa GPU-a. Bitno je za naglasiti da se za svaki podatak (piksel) organizira zasebna dretva, a na svim dretvama obavlja se ista naredba, čime je omogućen paralelizam izvođenja naredbi. GPU bolje savladava paralelnu obradu podataka od CPU-a. Za organizaciju resursa GPU-a, potrebno je razumjeti terminologiju: *grid*, *block* i *thread*.



Slika 5.1. Strukturalna organizacija arhitekture GPU-a

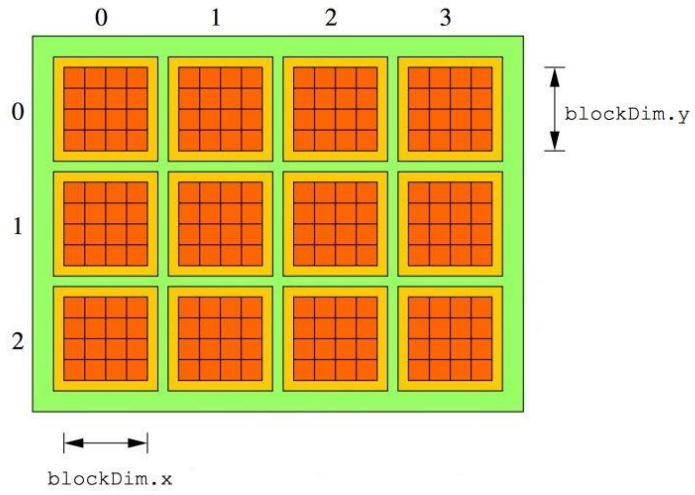
Grid se smatra superiornom organizacijskom cjelinom koji sadržava blokove. Poveznica među blokovima omogućena je globalnom memorijom, memorijom konstanti i memorijom za teksture. Globalna memorija je realizirana u sporijem DRAM-u, nije *cache*-irana, preko nje

pristupaju podacima za obradu svi blokovi unutar tog *grid*-a. Kako bi se omogućio pristup svim blokovima istog *grid*-a uvodi se sinkronizacijski mehanizam među samim blokovima. Memorija konstanti služi praktički za realizaciju filtra. Primjerice za linearne filtre i pripadnu jednadžbu oblika y=y1 + a*x, pohranjuju se koeficijenti u tu memoriju. Za navedenu grafičku karticu veličina je memorije konstanti je 64 kB. Memorija za teksture je isto realizirana sporijim DRAM-om, *cache*-irana je, podijeljena po gridu, i *read-only* (nema sekvence za upis u tu memoriju, a time je omogućena bolja kontrola nad istom). Grafičke primitive (trokuti, poligoni…) mogu se raditi i preko registara (kao i samo osjenčanje). Ukoliko to nije dovoljno koristi se lokalna memorija koja usporava procese, ali je zato realizirana dosta širokom sabirnicom, čime je omogućen paralelni prijenos podataka. Globalna memorija je dosta spora, procjenjuje se kašnjenje oko 100 ciklusa.

Grid je sastavljen od manjih organizacijskih podcjelina, blokova. Blok je sastavljen od *thread*-ova i njihovih pripadnih konteksta (skupa registara) i pripadnih lokalnih memorija, te zajedničke memorije za sve *thread*-ove unutar istog bloka. Veličina bloka u 1D je 32 (*thread*-a), što je određeno *warp size*-om. Maksimalan broj *thread*-ova po bloku je 1024, što je zapravo *warp size***warp size*. Dijeljena (*eng. shared*) memorija integrirana je na čipu i realizirana SRAM tehnologijom, odnosno omogućen joj je dosta brz pristup. Veličina za *NVIDIA Geforce GTX 850M* je 48 kB po bloku, a po multiprocesoru 64 kB. Česta je situacija u GPU arhitekturi da pri velikom broju ALU jedinica i registrima, nezgodno je da svaki *thread* ima zajedničku memoriju, stoga se za svaka 2 bloka dodaje dijeljena memorija namijenjena za napajanje s podacima. Svaki *thread* raspolaže i sa svojom sporijom lokalnom memorijom realiziranom u DRAM tehnici koja je zato relativno velika.

Najčešći problem koji se pojavljuje s radom dijeljene memorije je promašaj *cache*-a. Primjerice, pri obradi SIMD (*eng. Single Instruction Multiple Data*) od 8 skupa podataka (svaki podatak u jedan *thread*, a svi imaju istu naredbu), dogodi se slučaj da se jedan element vektora podataka iz *thread*-a ne nalazi u *cache*-u, nego u dijeljenoj memoriji. Taj problem rezultira dovodi do paralelizma izvođenja iste naredbe nad različitim podacima jer pristup dijeljenoj memoriji zahtijeva dodatno vrijeme od otprilike 20 ciklusa. Unutar tih 20 ciklusa, zbog nedostatka podataka sustav uspori ili čak stane (*eng. stall*). Rješenje tog problema je realiziracija preklapanja izvođenja brojnih fragmenata na jednoj jezgri kako bi se izbjegli navedeni zastoji.

U svaki *thread* upisuje se po jedan piksel podataka ulazne slike. Adresiranje piksela slike u CUDI obavlja se indeksima *col* i *row*.



Slika 5.2. Strukturalna organizacija jednog grida sastavljenog od blokova

Indeksi se izračunaju prema donje navedenim naredbama:

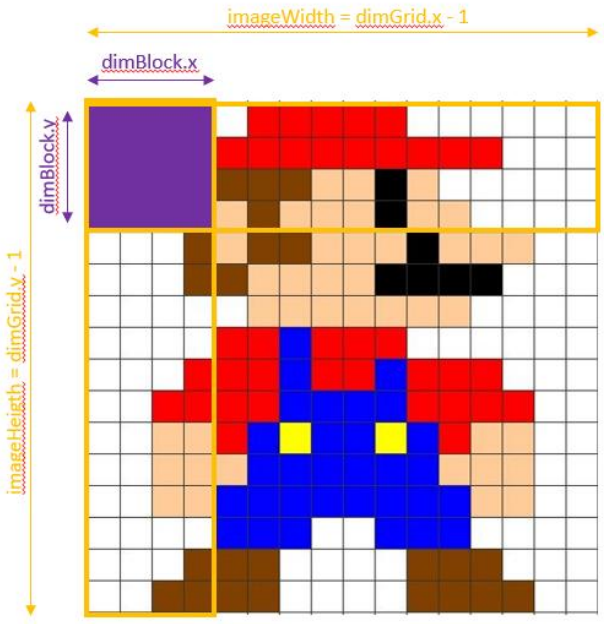
- col = blockIdx.x * blockDim.x + threadIdx.x
- row = blockIdx.y * blockDim.y + threadIdx.y

Time svaki piksel podataka ima svoj jedinstveni ID koji će poslužiti kao parametar za daljnju paralelnu obradu podataka median filtrom.

Najbitnije je za optimalan paralelizam realiziran pozivanjem CUDA *kernel*-a pronaći prikladan omjer veličine *grid*-a i veličine blokova s kojim će se pozivati navedeni *kernel*. Prvenstveno te brojke ovise o 1D dimenziji bloka te o osnovnim informacijama učitane slike (širina, visina, broj kanala).

Primjerice, učitana je slika veličine 16x16. Definiranje dimenzija *grid*-ova i blokova:

```
#define BLOCK 4
dim3 dimBlock(BLOCK, BLOCK); //(4*4*1)
dim3 dimGrid((int)ceil((float)imageWidth / (float)BLOCK),
(int)ceil((float)imageHeight / (float)BLOCK));
//(16/4)*(16/4)*1 = 4*4*1=16
```



Slika 5.3. Raspodjela piksela slike na određene grid-ove i pripadne blokove

Pri realizaciji median filtra dimenzije 3x3 CUDA-om treba se voditi računa o optimalnom odnosu veličine *grid*-a i bloka kojim se poziva *kernel*. Posebice je važno osigurati kritične dijelove (rubni) kako se ne bi učitavali podaci iz nedozvoljenih dijelova alocirane memorije na GPU. Stoga je u programskom kodu implementirano *if-else* grananje kojim se dopisuju nule u vanjskom okviru slike *zero padding* jer se pri izračunu medijana uzimaju u obzir i vrijednosti susjednih piksela, a na rubovima slike to je kritičan slučaj. Inicijalizacija indeksa *row* i *col*:

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

Provjera rubnih uvjeta i nadopuna nulama ciljane slike:

```
if ((row == 0) || (col == 0) || (row == imageHeight - 1) || (col == imageWidth - 1))
            outputImageKernel[row * imageWidth + col] = 0;
```

Za efikasnije postizanje efekta paralelizma prilikom izvršavanja implementiranog filtra korištena je dijeljena memorija, na način da je alocirana na kvadratični memorijski prostor dijeljene memorije GPU-a, širine i visine *WINDOW_SIZE*+2. Ovo dodavanje s 2 radi se zbog uzimanja okolnih piksela u obzir koji se nalaze izvan prozora, tj. okvir je popunjen nultim ili ne-nultim

vrijednostima (naknadno) kako bi se osigurao stabilan rad implementiranog filtra. Deklaracija takve memorije:

```
#define WINDOW_SIZE 3
__shared__ unsigned char sharedmem[(WINDOW_SIZE + 2)][(WINDOW_SIZE + 2)];
```

Popunjavanje dijeljene memorije s vrijednostima piksela slike u obradi obavlja na sljedeći način:

```
sharedmem[threadIdx.x + 1][threadIdx.y + 1] = inputImageKernel[row * imageWidth + col];
```

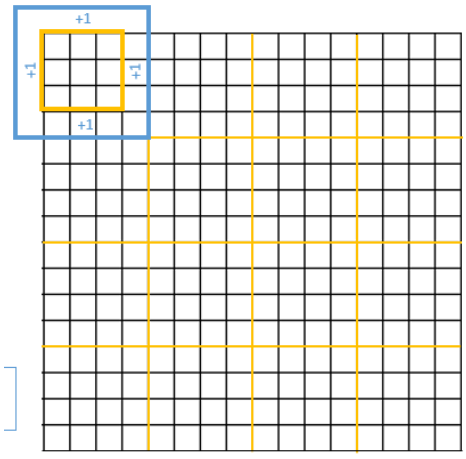
```
sharedmem[][] = imputImageKernel[rubni_dijelovi];
```

Kako bi se osigurala sinkronizacija *thread*-ova prilikom pristupanja podacima zajedničkoj memoriji i time sama konzistentnost podataka potrebno je voditi računa o CUDA naredbi:

```
__syncthreads();
```

Punjenje filter prozora s vrijednostima dijeljene memorije obavlja se naredbom:

```
unsigned char filterVector[9] =
{    sharedmem[threadIdx.x][threadIdx.y],    sharedmem[threadIdx.x    +    1][threadIdx.y],
sharedmem[threadIdx.x + 2][threadIdx.y],
   sharedmem[threadIdx.x][threadIdx.y + 1], sharedmem[threadIdx.x + 1][threadIdx.y + 1],
sharedmem[threadIdx.x + 2][threadIdx.y + 1],
   sharedmem[threadIdx.x][threadIdx.y + 2], sharedmem[threadIdx.x + 1][threadIdx.y + 2],
sharedmem[threadIdx.x + 2][threadIdx.y + 2] };
```



Slika 5.4. Plavi okvir predstavlja skener podataka za pohranu u zajedničku memoriju

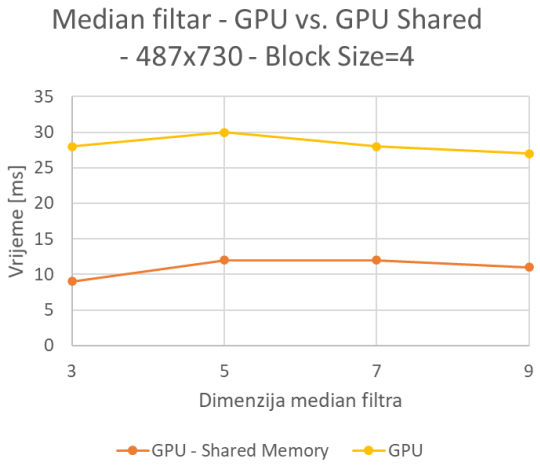
5.2. Analiza performansi

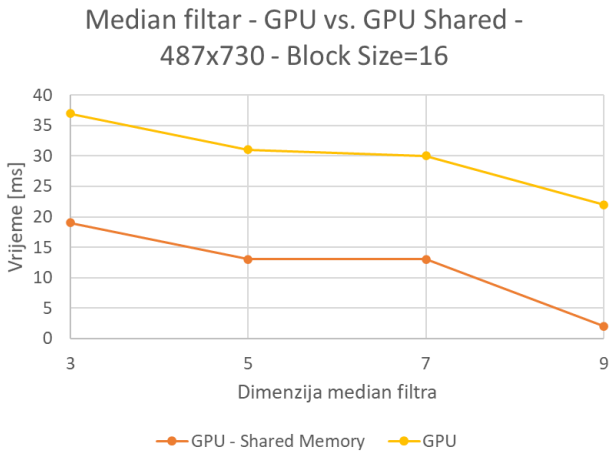
CPU vs. GPU Shared Memory vs. GPU:

FESB1CV.bmp BLOCK_SIZE = 4		Dimenzije median filtra			
		3x3	5x5	7x7	9x9
Hardware	CPU [ms]	47.997	315.657	1180.67	2607.65
	GPU [us] (Shared memory)	11	16	10	11
	GPU [us]	29	27	25	30

Prema izmjerenim vrijednostima vidljivo je da je za digitalnu obradu slike pogodniji GPU u odnosu na CPU. GPU je namijenjen za paralelno izvođenje naredbi, dok je CPU pogodan za sekvencijalno izvođenje naredbi (jedna za drugom). GPU ima više jezgri u odnosu na CPU, iako su one „slabije“.

Utjecaj dijeljene memorije i veličine bloka na rad GPU-a:

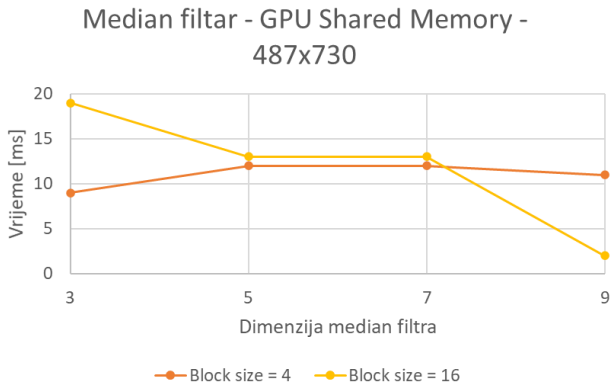




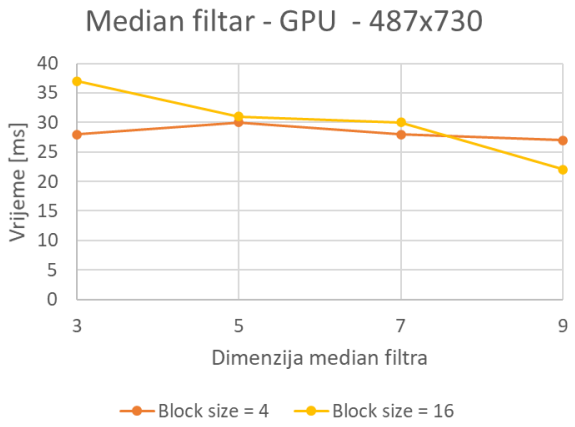
Dijeljena (*eng. shared*) memorija je realizirana na način da se u nju pohrane učitani pikseli s ulazne slike unutar površine *WINDOW_SIZE x WINDOW_SIZE*. Okvir se popuni nulama za ispravan rad algoritma filtra, te se preko indeksa *col* i *row* kvadratični prozor pomiče u desno duž reda. Kad prođe cijeli red koji je širine *imageWidth*, prelazi u novi red slike i ponavlja isti postupak.

Veličina dijeljene memorije ovisi o veličini prozora median filtra, a pomicanje o trenutnom *thread*-u unutar bloka koji je mapiran u 2D, duž x osi (*col*) te duž y osi (*row*). Očekivano je da je optimalno pokrenuti *kernel* s jednim blokom i više pripadnih dretvi, nego obratno. Veličina dijeljene memorije za *NVIDIA GTX 850M* je 48 kB, dretvi u bloku (1024, 1024, 64) i dimenzije *grid*-a (2147483647, 65535, 65535), *warp size* GPU je 32 (u kodu se ne smije premašiti #define BLOCK 32)

Za filtre manjih dimenzija sa strane performansi dobar je manji broj blokova.



Za filtre većih dimenzija sa strane performansi učinkovitija je obrada s većim brojem blokova.

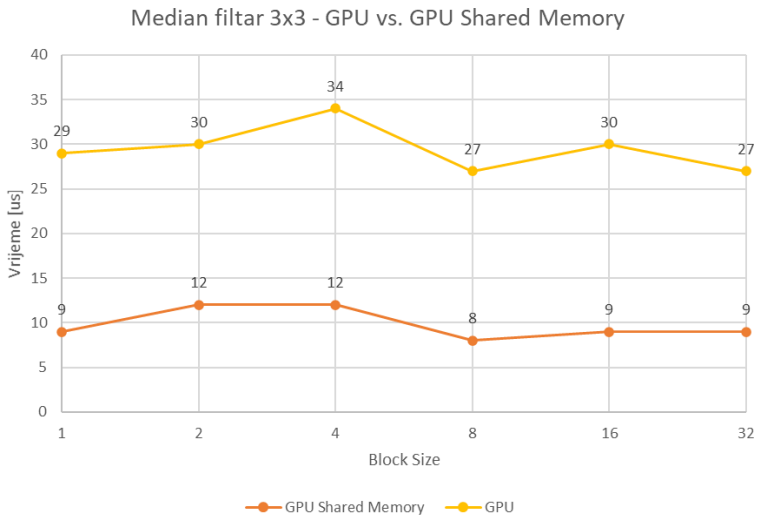


Utjecaj veličine blokova na rad filtra:

FESB1.jpg [us] (GPU-Shared memory)		Dimenzije median filtra				dimBlock	dimGrid
		3x3	5x5	7x7	9x9		
Block size	1	9	13	11	14	1	217800
	2	12	12	13	12	4	54450
	4	12	12	10	11	16	13613
	8	8	13	11	10	64	3403
	16	9	12	11	11	256	851
	32	9	13	12	10	1024	213

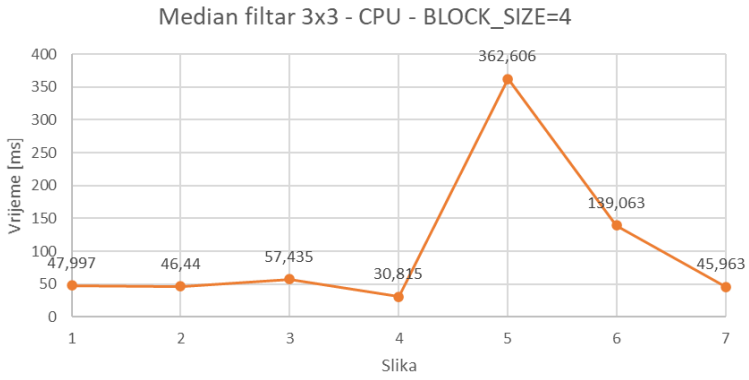
FESB4.jpg (GPU) [us]		Dimenzije median filtra				dimBlock	dimGrid
		3x3	5x5	7x7	9x9		
Block size	1	29	30	26	27	1	217800
	2	30	28	26	25	4	54450
	4	34	48	27	26	16	13613
	8	27	40	37	25	64	3403
	16	30	32	26	26	256	851
	32	27	28	27	25	1024	213

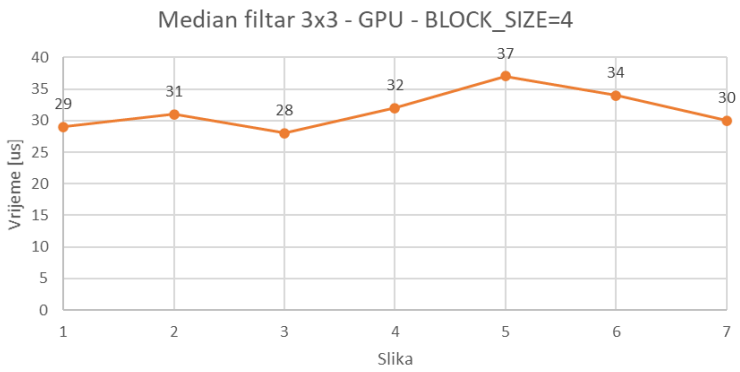
FESB4.jpg (CPU) [us]		Dimenzije median filtra			
		3x3	5x5	7x7	9x9
Block size	16	46.903	351.505	1185.33	2715.38

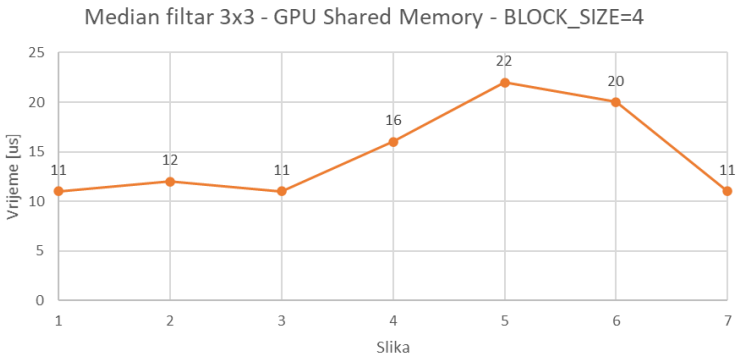


Utjecaj rezolucije slike na rad GPU-a:

Slika	1	FESB1CV.bmp (487x730)
	2	FESB2GS.bmp (500x750)
	3	FESB3GS.bmp (513x800)
	4	FESB4GS.bmp (330x660)
	5	SPLIT1GS.bmp (1053x2000)
	6	SPLIT2GS.bmp (900x1600)
	7	FESB1GS.bmp (487x730)







Bitno je za primijetiti da je vrijeme izvršavanja CPU-a reda veličine ms, a GPU-a reda veličine µs.

6. ZAKLJUČAK

Median filtar najdjelotvorniji je u uklanjanju *salt and pepper* šumova sa slike, što je demonstrirano ručnim dodavanjem šuma na slici koja je obrađena. Nadalje, korištenjem većih prozora filtra, bolje je potiskivanje šuma. Međutim, povećavanjem kvadratnog prozora, usporava se rad CPU ili GPU i javlja se sve jači efekt zamagljivanja slike (*eng. blur*). Iz priloženog koda, slike i rezultata obrade vidi se da je optimalan prozor 3x3 jer vrlo učinkovito obrađuje sliku, uz najmanje zamagljivanje i vrijeme izvršavanja.

Vrijeme izvršavanja prvenstveno ovisi o korištenoj arhitekturi procesora. Obradom slike pomoću CPU-a, GPU-a i GPU-a s dijeljenom memorijom može se zaključiti da je najbolje koristiti GPU s dijeljenom memorijom. Na performanse se može utjecati i promjenom veličine bloka u memoriji. Pokazano je da se povećanjem bloka smanjuje vrijeme izvršavanja, odnosno da je bolje pokrenuti *kernel* s jednim blokom i više pripadnih dretvi. To se najviše primjećuje kod filtara većih dimenzija prozora. Napravljena je i analiza vremena izvršavanja u slučaju promjene rezolucije slike. U sva tri slučaja (CPU, GPU i GPU s dijeljenom memorijom) očekivano je najveće vrijeme obrade slike s najvećom rezolucijom, a to je posebno naglašeno u slučaju izvođenja na CPU.

LITERATURA

[1] https://en.wikipedia.org/wiki/Image_noise

[2] https://homepages.inf.ed.ac.uk/rbf/HIPR2/median.htm

[3] https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/

[4] https://developer.nvidia.com/cuda-toolkit

[5] https://stackoverflow.com/questions/20186848/can-i-compile-a-cuda-program-without-having-a-cuda-device

[6] https://www.kickstarter.com/projects/opencv/opencv-for-beginners?ref=e1cqcw

[7] https://www.geeksforgeeks.org/opencv-overview/