NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY POLITEHNICA BUCHAREST
Faculty of Electronics, Telecommunications and Information Technology

Calcul paralel Project

Image Filtering Implementation in CUDA

Author: Francico Bessa Lopes Câmara

2024-2025

Contents

1. Project description

This project implements image filtering algorithms using NVIDIA CUDA to leverage GPU acceleration for efficient and scalable image processing. The objective is to demonstrate the performance improvement achieved by parallelizing convolution-based filtering operations, such as soft and sharp masking, using GPU resources compared to traditional CPU-based methods.

Purpose

Image filtering is a fundamental operation in image processing, used to enhance or extract features from images. Typical filters like soft (Gaussian blur) and sharp masks are computationally intensive, especially for large images or real-time applications. By utilizing CUDA, this project explores:

- Optimization techniques using global, shared and texture memory.
- Performance benchmarks to compare GPU implementations against each other and with CPU implementations.

2. Theoretical approach

Introduction

Image filtering is a mathematical operation that transforms an input image to enhance features, remove noise, or extract details. Convolution, the core operation in image filtering, is inherently parallelizable, as the output of each pixel can be computed independently. This makes it an ideal candidate for GPU acceleration using CUDA, where thousands of threads can operate simultaneously.

CUDA Architecture and Parallelism

CUDA (Compute Unified Device Architecture) provides a framework for developing GPU-accelerated applications. It introduces key abstractions such as threads, blocks, and grids, which allow developers to map computational tasks to hardware resources.

1. **Thread Hierarchy**:
    - **Threads**: Individual units of execution.
    - **Blocks**: Groups of threads that share resources like shared memory.
    - **Grids**: Collections of blocks that execute a kernel.
2. **Memory Types**:
    - **Global Memory**: Accessible by all threads but has high latency.

- **Shared Memory**: Faster, block-specific memory for communication between threads in the same block.
- **Texture Memory**: Read-only memory optimized for 2D spatial locality.

Image Filtering and Convolution

Convolution applies a filter kernel (matrix) to an image to produce an output image. The mathematical formulation is as follows:

$$G(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} F(x+i, y+j) \cdot H(i,j)$$

Where:

- G(x,y) is the output pixel intensity at (x, y).
- F(x+i,y+j) is the input pixel intensity in the neighbourhood defined by the kernel.
- H(i,j) is the kernel value.
- k is half the width of the kernel.

This operation is computationally expensive for large images or kernels, involving $O(M \cdot N \cdot K^2)$ computations for an M×N image and K×K kernel.

CUDA accelerates this operation by parallelizing the computation for each pixel.

Optimization Techniques

To improve the performance of the convolution operation, the following theoretical approaches are applied:

1. **Global Memory Access**:
   - Baseline implementation where each thread reads input data directly from global memory.
   - While functional, it suffers from high latency due to inefficient memory access patterns.
2. **Shared Memory Optimization**:
   - Shared memory is utilized to cache portions of the input image used by neighbouring threads.
   - Each thread block loads a tile of the image into shared memory, including a halo region for kernel overlap.
   - Reduces redundant global memory accesses and improves bandwidth utilization.
3. **Texture Memory Optimization**:
   - Texture memory is used to exploit spatial locality in 2D image data.

- Threads fetch pixel data from a texture bound to the input image, benefiting from the hardware's caching mechanism.
- Ideal for read-only data with a high degree of locality.

Parallelism in Filtering

Each thread in CUDA is assigned to compute the output for one pixel. For a filter kernel of size K×K:

- Threads collaborate within a block to preload image data into shared memory.
- Halo regions (extra rows/columns) are loaded to ensure kernel boundaries are handled correctly.
- Each thread computes its pixel value using the loaded data and the kernel.

3. Implementation and Practical Approach

Implementation

The project involves implementing image filtering algorithms using CUDA, focusing on different optimization strategies (global memory, shared memory and texture memory). The testing procedure aims to benchmark the algorithms on varying image resolutions and kernel sizes to understand their performance characteristics under different scenarios.

Practical Approach

1. **CPU Testing**:
    - **Objective**: Establish a baseline for comparison.
    - **Image Sizes**: Test with images of standard resolutions:
        - **VGA** (640x480), **SVGA** (800x600), **XGA** (1024x768)
        - **FHD** (1920x1080), **QHD** (2560x1440), **4K** (3840x2160)
        - **8K** (7680x4320), **10K** (10240x4320)
    - **Filter**: Use a fixed filter size (e.g., 3x3 Gaussian Blur) to measure performance.
    - **Benchmarking**: Measure execution time for each image size on the CPU.
2. **Simple CUDA Testing**:
    - **Objective**: Evaluate the performance of the global memory-based implementation.
    - **Image Sizes**: Test with images of standard resolutions:
        - **VGA** (640x480), **SVGA** (800x600), **XGA** (1024x768)
        - **FHD** (1920x1080), **QHD** (2560x1440), **4K** (3840x2160)

        - **8K** (7680x4320), **10K** (10240x4320)
    - **Kernel Sizes**: Test with kernels of increasing sizes:
        - 1x1, 2x2, 4x4, 8x8, 16x16, 32x32
    - **Benchmarking**: Measure kernel execution time for each kernel size.
3. **Shared Memory and Texture Memory Testing**:
    - **Objective**: Compare the optimized CUDA implementations (shared and texture memory).
    - **Image Sizes**: Test with images of standard resolutions:
        - **VGA** (640x480), **SVGA** (800x600), **XGA** (1024x768)
        - **FHD** (1920x1080), **QHD** (2560x1440), **4K** (3840x2160)
        - **8K** (7680x4320), **10K** (10240x4320)
    - **Kernel Sizes**: Test with kernels of increasing sizes:
        - 1x1, 2x2, 4x4, 8x8, 16x16, 32x32
    - **Benchmarking**: Measure execution time for each implementation, image size and kernel size.

Algorithm Description

Step 1: Data Preparation

- Load the input image and filter kernel into the GPU's global memory.
- If using texture memory, bind the image data to a texture reference.

Step 2: Thread Mapping

- Assign each thread a unique (x,y) coordinate corresponding to a pixel in the output image.
- Calculate thread indices using:
    - x = blockIdx.x · blockDim.x + threadIdx.x
    - y = blockIdx.y · blockDim.y + threadIdx.y

Step 3*: Shared Memory Optimization (if applicable)

- Allocate a shared memory array large enough to store the input data required by the block, including halo regions for kernel overlap.
- Load the required portion of the image into shared memory collaboratively:
    - Each thread loads a portion of the image.
    - Threads synchronize using ***__syncthreads()*** to ensure all data is loaded.

Step 3*: Texture Memory Optimization (if applicable)

- Use texture memory to access pixel values directly via 2D coordinates. The hardware caching mechanism reduces memory latency.

Step 4: Compute Convolution

- For each pixel, iterate over the filter kernel dimensions:
    - Multiply the corresponding input pixel value by the kernel value.
    - Sum the results to compute the output pixel value.
- Handle boundary conditions for pixels near the image edges.

Step 6: Store Results

- Write the computed pixel value to the output image in global memory.

Testing Workflow

1. **Setup**:
    - Ensure all required libraries (CUDA Toolkit, OpenCV) are installed.
    - Load test image of the specified resolutions.
2. **CPU Filtering**:
    - Implement a straightforward CPU-based convolution function.
    - Iterate over the list of test images, applying the filter and recording execution times.
3. **CUDA Filtering**:
    - Use three CUDA kernels:
        - **Global Memory Kernel**: Simple implementation without optimization.
        - **Shared Memory Kernel**: Optimized using shared memory.
        - **Texture Memory Kernel**: Optimized using texture memory.
    - For each test case:
        - Load the input image onto the GPU.
        - Configure appropriate grid and block sizes.
        - Run the kernel and measure execution time using CUDA events.
4. **Data Collection**:
    - Log execution times for all scenarios.
    - Collect results in a tabular format for analysis.

Tools and Libraries

1. **Hardware**: A CUDA-enabled GPU with sufficient memory to handle large images.
2. **Libraries**:
    - **CUDA Toolkit**: For GPU programming.
    - **OpenCV**: For image handling and preprocessing.

- **Standard C++ Libraries**: For file handling and performance measurement (<*chrono*> for CPU timing).
- **CUDA Events**: For GPU timing.

4. Measurements, Comparations, Experimental Results and Fine Tuning

Global Memory							
Time in ms		Kernel Size					
		1x1	2x2	4x4	8x8	16x16	32x32
Image size	VGA	3.74566	2.75442	2.63579	1.85959	1.51945	1.46946
	SVGA	5.6049	3.31996	2.99514	2.78092	2.60567	2.30219
	XGA	9.03179	5.08185	4.72343	4.3514	4.21735	3.737
	Full HD	19.56	11.1094	7.84357	6.67511	6.87096	6.72516
	QHD	35.7853	21.4864	12.8495	12.4551	11.5792	11.0928
	4k	78.173	40.1215	23.8773	22.6736	21.8215	21.1216
	8k	274.935	134.802	102.54	97.7562	95.9696	91.7761
	10k	361.506	176.416	133.182	123.702	121.646	115.502

Texture Memory							
Time in ms		Kernel Size					
		1x1	2x2	4x4	8x8	16x16	32x32
Image size	VGA	4.99994	4.47246	4.29188	4.14674	3.83134	3.71992
	SVGA	6.65827	6.11461	5.40132	5.01497	4.94998	4.60791
	XGA	10.4308	8.33468	8.9222	7.41674	7.35305	7.19038
	Full HD	23.859	18.2668	15.3026	14.2183	13.7347	13.653
	QHD	39.2184	30.181	28.445	27.2594	26.1852	25.141
	4k	71.2867	59.4063	58.5288	57.6178	56.0287	54.5244
	8k	282.493	187.948	192.563	189.493	187.226	175.809
	10k	360.728	283.676	280.847	242.973	236.78	232.513

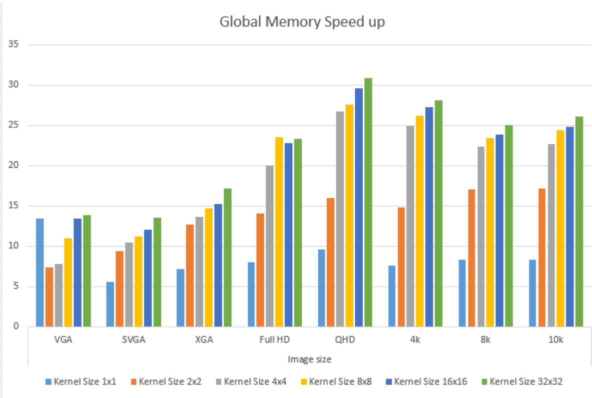
Shared Memory							
Time in ms		Kernel Size					
		1x1	2x2	4x4	8x8	16x16	32x32
Image size	VGA	4.57792	2.16445	1.75268	1.65587	1.48879	1.38456
	SVGA	5.92259	3.47828	2.42491	2.3313	2.19706	1.77143
	XGA	10.0885	4.77145	4.50009	3.68183	3.6183	3.49198
	Full HD	22.0513	12.0255	7.60879	6.83679	6.7633	6.39565
	QHD	39.3759	19.5172	12.579	11.6736	11.5475	11.3699
	4k	80.7283	40.1215	25.0734	22.911	22.4407	21.7839
	8k	307.652	133.929	91.5479	79.258	82.3819	80.0368
	10k	408.469	179.962	115.142	109.848	100.399	98.637

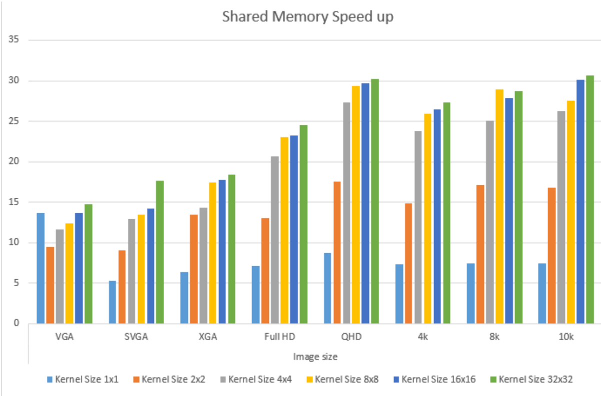
CPU (in ms)		
Image size	VGA	20.4366
	SVGA	31.3131
	XGA	64.2586
	Full HD	157.015
	QHD	343.186
	4k	594.818
	8k	2293.75
	10k	3019.67

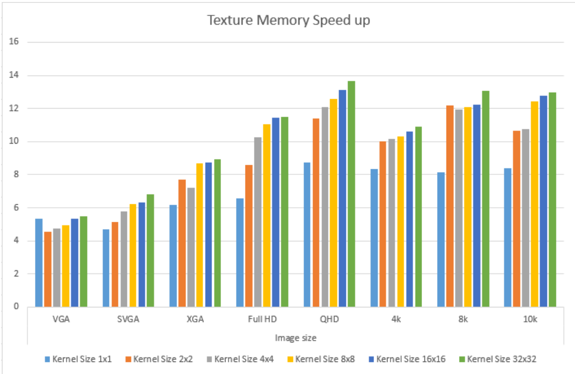
Global Memory Speed up							
		Kernel Size					
		1x1	2x2	4x4	8x8	16x16	32x32
Image size	VGA	13.45	7.419566	7.753501	10.98984	13.45	13.90756
	SVGA	5.586737	9.43177	10.45464	11.25998	12.01729	13.60144
	XGA	7.114714	12.64473	13.60422	14.76734	15.23672	17.19524
	Full HD	8.027352	14.13353	20.01831	23.52246	22.85197	23.3474
	QHD	9.590139	15.97224	26.70812	27.55385	29.63814	30.93773
	4k	7.608995	14.82542	24.91144	26.23395	27.25835	28.1616
	8k	8.342881	17.0157	22.36932	23.46398	23.9008	24.99289
	10k	8.353029	17.11676	22.67326	24.41084	24.82342	26.14388

Texture Memory Speed up							
		Kernel Size					
		1x1	2x2	4x4	8x8	16x16	32x32
Image size	VGA	5.334061	4.569432	4.76169	4.928353	5.334061	5.493828
	SVGA	4.702888	5.12103	5.797305	6.243926	6.325904	6.79551
	XGA	6.160467	7.709786	7.202103	8.663995	8.73904	8.936746
	Full HD	6.580955	8.595649	10.26067	11.04316	11.43199	11.5004
	QHD	8.750637	11.37093	12.0649	12.58964	13.10611	13.65045
	4k	8.344025	10.01271	10.16283	10.32351	10.61631	10.90921
	8k	8.11967	12.20417	11.91169	12.10467	12.25124	13.04683
	10k	8.371044	10.64478	10.75201	12.42801	12.75306	12.9871

Shared Memory Speed up							
		Kernel Size					
		1x1	2x2	4x4	8x8	16x16	32x32
Image size	VGA	13.72699	9.441937	11.6602	12.34191	13.72699	14.76036
	SVGA	5.287062	9.002467	12.9131	13.4316	14.25227	17.67674
	XGA	6.36949	13.46731	14.2794	17.4529	17.75933	18.40177
	Full HD	7.120442	13.05684	20.636	22.96619	23.21574	24.55028
	QHD	8.715636	17.58377	27.28245	29.39847	29.71951	30.18373
	4k	7.368147	14.82542	23.72307	25.96211	26.50621	27.3054
	8k	7.455664	17.12661	25.05519	28.9403	27.84289	28.65869
	10k	7.392654	16.77949	26.22562	27.48953	30.07669	30.61397







The performance benchmarks for the implemented image filtering algorithms were obtained using an NVIDIA RTX 3080 laptop GPU. The experiments evaluated the execution time of convolution-based image filtering under various configurations, including different memory optimization strategies (global memory, shared memory, and texture memory), image resolutions, and kernel sizes.

Observations from Graphs

1. **Global Memory Performance**:
    - The baseline implementation utilizing global memory displayed significantly higher execution times, particularly for larger images and kernel sizes. This was expected due to the high latency and inefficient memory access patterns associated with global memory.
2. **Shared Memory Optimization**:
    - Shared memory utilization showed a dramatic improvement in execution times, particularly for higher-resolution images. The reduced global memory accesses, coupled with the efficient caching of image tiles in shared memory, allowed for better bandwidth utilization.
    - The impact of shared memory optimization was especially noticeable for structured access patterns, such as convolution operations with larger kernels (e.g., 32x32).

3. **Texture Memory Optimization**:
    - While texture memory did not outperform shared memory in scenarios with regular access patterns, it excelled in cases with irregular or scattered access patterns due to its hardware-accelerated 2D spatial locality optimization.
    - The results demonstrated texture memory's strength in interpolation and its ability to handle non-coalesced access efficiently. However, the overhead introduced by normalization and data conversion limited its performance for dense convolution tasks.
4. **Scaling with Image Resolution**:
    - As the resolution increased, all implementations exhibited longer execution times, with the global memory implementation scaling poorly. Both shared and texture memory implementations showed much better scalability, particularly when the resolution exceeded Full HD (1920x1080).
5. **Kernel Size Impact**:
    - The shared memory implementation maintained relatively consistent performance improvements across varying kernel sizes, thanks to its ability to load and reuse image tiles effectively.
    - Texture memory's performance was more sensitive to kernel size, as larger kernels increased the complexity of interpolation and reduced the effectiveness of caching mechanisms.

Comparative Performance Analysis

Texture memory performed worse in structured convolution tasks due to its fixed-function design, which introduces overhead from normalization and interpolation even when not needed. While optimized for irregular access patterns, its caching mechanism struggles with dense, regular accesses, leading to inefficiencies. Additionally, preprocessing data for texture memory and managing halo regions for large kernels add significant overhead. Unlike shared memory, texture memory lacks explicit control, limiting its suitability for tasks requiring fine-grained optimization, as seen in this implementation.

The graphs clearly illustrate the superior performance of shared memory optimization over both global and texture memory for structured convolution tasks. For the largest image size tested (10K resolution, 10240x4320 pixels), the shared memory implementation outperformed global memory and texture, highlighting the effectiveness of manual memory management in CUDA.

5. Conclusions

This project successfully implemented and analysed CUDA-based image filtering algorithms on an NVIDIA RTX 3080 laptop GPU. By comparing global memory, shared memory, and texture memory strategies, we evaluated their effectiveness for convolution tasks.

The results demonstrated that shared memory optimization provided the best performance for structured and regular access patterns, leveraging explicit caching and reduced memory latency. Global memory, while less optimized than shared memory, outperformed texture memory in dense convolution tasks. This was due to the latter's overheads from normalization, interpolation, and limited adaptability to structured access patterns, which reduced its efficiency.

The study highlights the importance of aligning memory strategies with task characteristics for optimal performance in GPU programming. CUDA's parallelism and flexibility offer significant speed ups over traditional CPU implementations, particularly when memory is efficiently managed.

6. Annexes (implemented code)

```
#include <iostream>
#include <cuda_runtime.h>
#include <opencv2/opencv.hpp>
#include <chrono> // For CPU benchmarking

using namespace std;
using namespace cv;

static int kernel = 1;  // Kernel size for CUDA kernel, change for different kernel
sizes
static string PATH = ".\\Images\\XGA.jpg"; // Path to the image, change for other
images

// CUDA kernel using global memory
__global__ void applyFilter(const unsigned char* input, unsigned char* output, int
width, int height, int channels, const float* filter, int filterWidth) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= width || y >= height) return;

    int halfFilterWidth = filterWidth / 2;
    float pixelSum[3] = { 0.0f, 0.0f, 0.0f };

    for (int ky = -halfFilterWidth; ky <= halfFilterWidth; ++ky) {
        for (int kx = -halfFilterWidth; kx <= halfFilterWidth; ++kx) {
            int imgX = min(max(x + kx, 0), width - 1);
            int imgY = min(max(y + ky, 0), height - 1);

            int imgIdx = (imgY * width + imgX) * channels;
            int filterIdx = (ky + halfFilterWidth) * filterWidth + (kx +
halfFilterWidth);

            for (int c = 0; c < channels; ++c) {
                pixelSum[c] += input[imgIdx + c] * filter[filterIdx];
            }
        }
    }

    int outputIdx = (y * width + x) * channels;
    for (int c = 0; c < channels; ++c) {
        output[outputIdx + c] = min(max(int(pixelSum[c]), 0), 255);
    }
}

void applyCUDAFilter(const Mat& inputImage, Mat& outputImage, const float* filter,
int filterWidth) {
    int width = inputImage.cols;
    int height = inputImage.rows;
    int channels = inputImage.channels();
    size_t imageSize = width * height * channels * sizeof(unsigned char);
    size_t filterSize = filterWidth * filterWidth * sizeof(float);

    unsigned char* d_input, * d_output;
    float* d_filter;

    cudaMalloc((void**)&d_input, imageSize);
    cudaMalloc((void**)&d_output, imageSize);
    cudaMalloc((void**)&d_filter, filterSize);

    cudaMemcpy(d_input, inputImage.data, imageSize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_filter, filter, filterSize, cudaMemcpyHostToDevice);

    dim3 blockSize(kernel, kernel);
```

```
    dim3 gridSize((width + blockSize.x - 1) / blockSize.x, (height + blockSize.y -
1) / blockSize.y);

    applyFilter << <gridSize, blockSize >> > (d_input, d_output, width, height,
channels, d_filter, filterWidth);

    cudaDeviceSynchronize();
    cudaMemcpy(outputImage.data, d_output, imageSize, cudaMemcpyDeviceToHost);

    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_filter);
}

// CUDA kernel using shared memory
__global__ void applyFilterShared(const unsigned char* input, unsigned char*
output, int width, int height, int channels, const float* filter, int filterWidth)
{
    extern __shared__ unsigned char sharedMem[];

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    int localX = threadIdx.x;
    int localY = threadIdx.y;

    int halfFilterWidth = filterWidth / 2;
    int sharedWidth = blockDim.x + 2 * halfFilterWidth;
    int sharedHeight = blockDim.y + 2 * halfFilterWidth;

    int sharedIdx = ((localY + halfFilterWidth) * sharedWidth + (localX +
halfFilterWidth)) * channels;

    int globalIdx = (y * width + x) * channels;

    for (int c = 0; c < channels; ++c) {
        if (x < width && y < height) {
            sharedMem[sharedIdx + c] = input[globalIdx + c];
        }
        else {
            sharedMem[sharedIdx + c] = 0;
        }
    }

    for (int c = 0; c < channels; ++c) {
        // Left halo
        if (localX < halfFilterWidth) {
            int sharedHaloIdx = ((localY + halfFilterWidth) * sharedWidth + localX)
* channels + c;
            int imgX = max(x - halfFilterWidth, 0);
            sharedMem[sharedHaloIdx] = input[(y * width + imgX) * channels + c];
        }

        // Right halo
        if (localX >= blockDim.x - halfFilterWidth) {
            int sharedHaloIdx = ((localY + halfFilterWidth) * sharedWidth + (localX
+ 2 * halfFilterWidth)) * channels + c;
            int imgX = min(x + halfFilterWidth, width - 1);
            sharedMem[sharedHaloIdx] = input[(y * width + imgX) * channels + c];
        }

        // Top halo
        if (localY < halfFilterWidth) {
            int sharedHaloIdx = (localY * sharedWidth + (localX + halfFilterWidth))
* channels + c;
            int imgY = max(y - halfFilterWidth, 0);
            sharedMem[sharedHaloIdx] = input[(imgY * width + x) * channels + c];
        }
```

```
        // Bottom halo
        if (localY >= blockDim.y - halfFilterWidth) {
            int sharedHaloIdx = ((localY + 2 * halfFilterWidth) * sharedWidth +
(localX + halfFilterWidth)) * channels + c;
            int imgY = min(y + halfFilterWidth, height - 1);
            sharedMem[sharedHaloIdx] = input[(imgY * width + x) * channels + c];
        }
    }

    __syncthreads();

    if (x < width && y < height) {
        float pixelSum[3] = { 0.0f, 0.0f, 0.0f };

        for (int ky = -halfFilterWidth; ky <= halfFilterWidth; ++ky) {
            for (int kx = -halfFilterWidth; kx <= halfFilterWidth; ++kx) {
                int sharedConvIdx = ((localY + halfFilterWidth + ky) * sharedWidth
+ (localX + halfFilterWidth + kx)) * channels;
                int filterIdx = (ky + halfFilterWidth) * filterWidth + (kx +
halfFilterWidth);

                for (int c = 0; c < channels; ++c) {
                    pixelSum[c] += sharedMem[sharedConvIdx + c] *
filter[filterIdx];
                }
            }
        }

        for (int c = 0; c < channels; ++c) {
            output[globalIdx + c] = min(max(int(pixelSum[c]), 0), 255);
        }
    }
}

void applyCUDAFilterShared(const Mat& inputImage, Mat& outputImage, const float*
filter, int filterWidth) {
    int width = inputImage.cols;
    int height = inputImage.rows;
    int channels = inputImage.channels();
    size_t imageSize = width * height * channels * sizeof(unsigned char);
    size_t filterSize = filterWidth * filterWidth * sizeof(float);

    unsigned char* d_input, * d_output;
    float* d_filter;

    cudaMalloc((void**)&d_input, imageSize);
    cudaMalloc((void**)&d_output, imageSize);
    cudaMalloc((void**)&d_filter, filterSize);

    cudaMemcpy(d_input, inputImage.data, imageSize, cudaMemcpyHostToDevice);
    cudaMemcpy(d_filter, filter, filterSize, cudaMemcpyHostToDevice);

    dim3 blockSize(kernel, kernel);
    dim3 gridSize((width + blockSize.x - 1) / blockSize.x, (height + blockSize.y -
1) / blockSize.y);

    int sharedMemSize = (blockSize.x + filterWidth - 1) * (blockSize.y +
filterWidth - 1) * channels * sizeof(unsigned char);
    applyFilterShared << <gridSize, blockSize, sharedMemSize >> > (d_input,
d_output, width, height, channels, d_filter, filterWidth);

    cudaMemcpy(outputImage.data, d_output, imageSize, cudaMemcpyDeviceToHost);

    cudaFree(d_input);
    cudaFree(d_output);
    cudaFree(d_filter);
}
```

```
// CUDA kernel using texture memory
__global__ void applyFilterTexture(cudaTextureObject_t texObj, unsigned char*
output, int width, int height, int channels, const float* filter, int filterWidth)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= width || y >= height) return;

    int halfFilterWidth = filterWidth / 2;
    float pixelSum[4] = { 0.0f, 0.0f, 0.0f, 0.0f };

    for (int ky = -halfFilterWidth; ky <= halfFilterWidth; ++ky) {
        for (int kx = -halfFilterWidth; kx <= halfFilterWidth; ++kx) {
            int filterIdx = (ky + halfFilterWidth) * filterWidth + (kx +
halfFilterWidth);

            float4 texValue = tex2D<float4>(texObj, x + kx + 0.5f, y + ky + 0.5f);

            pixelSum[0] += texValue.x * filter[filterIdx]; // R
            pixelSum[1] += texValue.y * filter[filterIdx]; // G
            pixelSum[2] += texValue.z * filter[filterIdx]; // B
            if (channels == 4) {
                pixelSum[3] += texValue.w * filter[filterIdx]; // A
            }
        }
    }

    int outputIdx = (y * width + x) * channels;
    output[outputIdx] = min(max(int(pixelSum[0] * 255.0f), 0), 255); // R
    output[outputIdx + 1] = min(max(int(pixelSum[1] * 255.0f), 0), 255); // G
    output[outputIdx + 2] = min(max(int(pixelSum[2] * 255.0f), 0), 255); // B
    if (channels == 4) {
        output[outputIdx + 3] = min(max(int(pixelSum[3] * 255.0f), 0), 255); // A
    }
}

void applyCUDAFilterTexture(const Mat& inputImage, Mat& outputImage, const float*
filter, int filterWidth) {
    int width = inputImage.cols;
    int height = inputImage.rows;
    int channels = inputImage.channels();

    Mat formattedInput;
    if (channels == 3) {
        cvtColor(inputImage, formattedInput, COLOR_BGR2BGRA);
        channels = 4;
    }
    else {
        formattedInput = inputImage.clone();
    }

    size_t imageSize = width * height * channels * sizeof(unsigned char);
    size_t filterSize = filterWidth * filterWidth * sizeof(float);

    unsigned char* d_output;
    float* d_filter;

    cudaMalloc((void**)&d_output, imageSize);
    cudaMalloc((void**)&d_filter, filterSize);

    cudaMemcpy(d_filter, filter, filterSize, cudaMemcpyHostToDevice);

    cudaChannelFormatDesc channelDesc = cudaCreateChannelDesc<uchar4>();
    cudaArray* cuArray;
    cudaMallocArray(&cuArray, &channelDesc, width, height);
```

```
    cudaMemcpy2DToArray(cuArray, 0, 0, formattedInput.ptr(), formattedInput.step,
formattedInput.step, height, cudaMemcpyHostToDevice);

    cudaResourceDesc resDesc = {};
    resDesc.resType = cudaResourceTypeArray;
    resDesc.res.array.array = cuArray;

    cudaTextureDesc texDesc = {};
    texDesc.addressMode[0] = cudaAddressModeClamp;
    texDesc.addressMode[1] = cudaAddressModeClamp;
    texDesc.filterMode = cudaFilterModePoint;
    texDesc.readMode = cudaReadModeNormalizedFloat;
    texDesc.normalizedCoords = false;

    cudaTextureObject_t texObj = 0;
    cudaCreateTextureObject(&texObj, &resDesc, &texDesc, nullptr);

    dim3 blockSize(kernel, kernel);
    dim3 gridSize((width + blockSize.x - 1) / blockSize.x, (height + blockSize.y -
1) / blockSize.y);

    applyFilterTexture << <gridSize, blockSize >> > (texObj, d_output, width,
height, channels, d_filter, filterWidth);

    if (channels == 4) {
        Mat rgbaOutput(height, width, CV_8UC4);
        cudaMemcpy(rgbaOutput.data, d_output, imageSize, cudaMemcpyDeviceToHost);

        cvtColor(rgbaOutput, outputImage, COLOR_BGRA2BGR);
    }
    else {
        cudaMemcpy(outputImage.data, d_output, imageSize, cudaMemcpyDeviceToHost);
    }

    cudaDestroyTextureObject(texObj);
    cudaFreeArray(cuArray);
    cudaFree(d_output);
    cudaFree(d_filter);
}

// CPU implementation
void benchmarkCPU(const Mat& inputImage, Mat& outputImage, const float* filter, int
filterWidth) {
    int width = inputImage.cols;
    int height = inputImage.rows;
    int channels = inputImage.channels();
    int halfFilterWidth = filterWidth / 2;

    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            float pixelSum[3] = { 0.0f, 0.0f, 0.0f };
            for (int ky = -halfFilterWidth; ky <= halfFilterWidth; ++ky) {
                for (int kx = -halfFilterWidth; kx <= halfFilterWidth; ++kx) {
                    int imgX = min(max(x + kx, 0), width - 1);
                    int imgY = min(max(y + ky, 0), height - 1);

                    for (int c = 0; c < channels; ++c) {
                        pixelSum[c] += inputImage.at<Vec3b>(imgY, imgX)[c] *
filter[(ky + halfFilterWidth) * filterWidth + (kx + halfFilterWidth)];
                    }
                }
            }
            for (int c = 0; c < channels; ++c) {
                outputImage.at<Vec3b>(y, x)[c] = min(max(int(pixelSum[c]), 0),
255);
            }
        }
    }
```

```
}

int main() {
    string imagePath = PATH;
    Mat inputImage = imread(imagePath, IMREAD_COLOR);

    if (inputImage.empty()) {
        cerr << "Error: Could not load image." << endl;
        return -1;
    }

    Mat outputImageCUDA_Global_soft = inputImage.clone();
    Mat outputImageCUDA_Global_sharp = inputImage.clone();
    Mat outputImageCUDA_Shared_soft = inputImage.clone();
    Mat outputImageCUDA_Shared_sharp = inputImage.clone();
    Mat outputImageCUDA_Texture_soft = inputImage.clone();
    Mat outputImageCUDA_Texture_sharp = inputImage.clone();
    Mat outputImageCPU_soft = inputImage.clone();
    Mat outputImageCPU_sharp = inputImage.clone();

    float softFilter[] = {
        1 / 16.0f, 2 / 16.0f, 1 / 16.0f,
        2 / 16.0f, 4 / 16.0f, 2 / 16.0f,
        1 / 16.0f, 2 / 16.0f, 1 / 16.0f
    };

    float sharpFilter[] = {
       0, -1, 0,
       -1, 5, -1,
       0, -1, 0
    };

    int filterWidth = 3;

    // Timing GPU execution Global Memory
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    cout << "Applying CUDA filter with global memory..." << endl;
    cudaEventRecord(start);
    for (int i = 0; i < 10; i++)
    {
        applyCUDAFilter(inputImage, outputImageCUDA_Global_soft, softFilter,
filterWidth);
        applyCUDAFilter(inputImage, outputImageCUDA_Global_sharp, sharpFilter,
filterWidth);
    }
    cudaEventRecord(stop);
    cudaDeviceSynchronize();
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    cout << "CUDA Kernel Execution Time using Global memory: " << milliseconds/10
<< " ms" << endl;

    // Timing GPU execution Shared Memory
    cout << "Applying CUDA filter with shared memory..." << endl;
    cudaEventRecord(start);
    for (int i = 0; i < 10; i++)
    {
        applyCUDAFilterShared(inputImage, outputImageCUDA_Shared_soft, softFilter,
filterWidth);
        applyCUDAFilterShared(inputImage, outputImageCUDA_Shared_sharp,
sharpFilter, filterWidth);
    }
    cudaEventRecord(stop);
```

```
    cudaDeviceSynchronize();
     milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    cout << "CUDA Kernel Execution Time using Shared memory: " << milliseconds/10
<< " ms" << endl;

    // Timing GPU execution Texture Memory
    cout << "Applying CUDA filter with texture memory..." << endl;
    cudaEventRecord(start);
    for (int i = 0; i < 10; i++)
    {
        applyCUDAFilterTexture(inputImage, outputImageCUDA_Texture_soft,
softFilter, filterWidth);
        applyCUDAFilterTexture(inputImage, outputImageCUDA_Texture_sharp,
sharpFilter, filterWidth);
    }
    cudaEventRecord(stop);
    cudaDeviceSynchronize();
     milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    cout << "CUDA Kernel Execution Time using Texture memory: " << milliseconds/10
<< " ms" << endl;

    //Timing CPU execution
    cout << "Benchmarking CPU..." << endl;
    auto start_CPU = chrono::high_resolution_clock::now();
    benchmarkCPU(inputImage, outputImageCPU_soft, softFilter, filterWidth);
    benchmarkCPU(inputImage, outputImageCPU_sharp, sharpFilter, filterWidth);
    auto stop_CPU = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> elapsed = stop_CPU - start_CPU;
    cout << "CPU Execution Time: " << elapsed.count() << " ms" << endl;

    imshow("Input Image", inputImage);
    imshow("Filtered Image CUDA Soft Filter using Global memory",
outputImageCUDA_Global_soft);
    imshow("Filtered Image CUDA Sharp Filter using Global memory",
outputImageCUDA_Global_sharp);
    imshow("Filtered Image CUDA Soft Filter using Shared memory",
outputImageCUDA_Shared_soft);
    imshow("Filtered Image CUDA Sharp Filter using Shared memory",
outputImageCUDA_Shared_sharp);
    imshow("Filtered Image CUDA Soft Filter using Texture memory",
outputImageCUDA_Texture_soft);
    imshow("Filtered Image CUDA Sharp Filter using Texture memory",
outputImageCUDA_Texture_sharp);
    imshow("Filtered Image CPU Soft Filter", outputImageCPU_soft);
    imshow("Filtered Image CPU Sharp Filter", outputImageCPU_sharp);
    waitKey(0);

    return 0;
}
```