gpu code

```
%%writefile programs/meanFilter.cu
// CUDA implementation of Mean Filter
#include "opencv2/imgproc/imgproc.hpp"
#include <opencv2/highgui.hpp>
#include <iostream>
#include <string>
#include <stdio.h>
#include <cuda.h>
#include "cuda_runtime.h"

#define BLOCK_SIZE      16
#define FILTER_WIDTH    8
#define FILTER_HEIGHT   8

using namespace std;

// Run Mean Filter on GPU
__global__ void meanFilter(unsigned char *srcImage, unsigned char *dstImage, unsigned int width,
unsigned int height, int channel)
{
  int x = blockIdx.x*blockDim.x + threadIdx.x;
  int y = blockIdx.y*blockDim.y + threadIdx.y;

  // only threads inside image will write results
  if((x>=FILTER_WIDTH/2) && (x<(width-FILTER_WIDTH/2)) && (y>=FILTER_HEIGHT/2) &&
(y<(height-FILTER_HEIGHT/2)))
  {
    for(int c=0 ; c<channel ; c++)
    {
      unsigned char filterVector[FILTER_WIDTH*FILTER_HEIGHT];
      int sum = 0;
      // Loop inside the filter to sum pixel values
      for(int ky=0; ky<FILTER_HEIGHT; ky++) {
    for(int kx=0; kx<FILTER_WIDTH; kx++) {
       filterVector[ky*FILTER_WIDTH+kx] = srcImage[((y+ky-FILTER_HEIGHT/2)*width +
(x+kx-FILTER_WIDTH/2))*channel+c];
       sum += filterVector[ky*FILTER_WIDTH+kx];
    }
 }

      // Calculate mean value
      dstImage[(y*width+x)*channel+c] = sum / (FILTER_WIDTH*FILTER_HEIGHT);
    }
  }
}

// The wrapper to call mean filter
extern "C" void meanFilter_GPU_wrapper(const cv::Mat& input, cv::Mat& output)
{
     // Use cuda event to catch time
     cudaEvent_t start, stop;
     cudaEventCreate(&start);
     cudaEventCreate(&stop);
     // Calculate number of image channels
     int channel = input.step/input.cols;

     // Calculate number of input & output bytes in each block
     const int inputSize = input.cols * input.rows * channel;
     const int outputSize = output.cols * output.rows * channel;
     unsigned char *d_input, *d_output;

     // Allocate device memory
     cudaMalloc<unsigned char>(&d_input,inputSize);
     cudaMalloc<unsigned char>(&d_output,outputSize);

     // Copy data from OpenCV input image to device memory
     cudaMemcpy(d_input,input.ptr(),inputSize,cudaMemcpyHostToDevice);

     // Specify block size
     const dim3 block(BLOCK_SIZE,BLOCK_SIZE);

     // Calculate grid size to cover the whole image
     const dim3 grid((output.cols + block.x - 1)/block.x, (output.rows + block.y - 1)/block.y);

     // Start time
     cudaEventRecord(start);

     // Run Mean Filter kernel on CUDA
     meanFilter<<<grid,block>>>(d_input, d_output, output.cols, output.rows, channel);

     // Stop time
     cudaEventRecord(stop);

     //Copy data from device memory to output image
     cudaMemcpy(output.ptr(),d_output,outputSize,cudaMemcpyDeviceToHost);

     //Free the device memory
     cudaFree(d_input);
     cudaFree(d_output);

     cudaEventSynchronize(stop);
     float milliseconds = 0;

     // Calculate elapsed time in milisecond
     cudaEventElapsedTime(&milliseconds, start, stop);
     cout<< "\nProcessing time on GPU for mean filtering: " << milliseconds << "\n";
}
```

```
%%writefile programs/dilation.cu

#include "opencv2/imgproc/imgproc.hpp"
#include <opencv2/highgui.hpp>
#include <iostream>
#include <string>
#include <stdio.h>
```

```
#include <cuda.h>
#include "cuda_runtime.h"

#define BLOCK_SIZE      16
#define FILTER_WIDTH    8
#define FILTER_HEIGHT   8

using namespace std;


__global__ void dilationFilter(unsigned char *srcImage, unsigned char *dstImage, unsigned int width,
unsigned int height, int channel)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Check if the thread is within the image boundaries
    if (x < width && y < height)
    {
        for (int c = 0; c < channel; c++)
        {
            unsigned char maxPixel = 0; // Initialize maxPixel to the minimum possible value

            // Loop through the filter window
            for (int ky = -FILTER_HEIGHT / 2; ky <= FILTER_HEIGHT / 2; ky++)
            {
                for (int kx = -FILTER_WIDTH / 2; kx <= FILTER_WIDTH / 2; kx++)
                {
                    // Calculate the neighbor’s coordinates
                    int nx = x + kx;
                    int ny = y + ky;

                    // Check if the neighbor is within the image boundaries
                    if (nx >= 0 && nx < width && ny >= 0 && ny < height)
                    {
                        // Get the neighbor’s pixel value
                        unsigned char neighborPixel = srcImage[(ny * width + nx) * channel + c];

                        // Update maxPixel if the neighbor’s pixel value is greater
                        if (neighborPixel > maxPixel)
                        {
                            maxPixel = neighborPixel;
                        }
                    }
                }
            }

            // Set the dilation result
            dstImage[(y * width + x) * channel + c] = maxPixel;
        }
    }
}
```

```
extern "C" void dilation_GPU_wrapper(const cv::Mat& input, cv::Mat& output)
{
	// Use cuda event to catch time
	cudaEvent_t start, stop;
	cudaEventCreate(&start);
	cudaEventCreate(&stop);

	// Calculate number of image channels
	int channel = input.step/input.cols;

	// Calculate number of input & output bytes in each block
	const int inputSize = input.cols * input.rows * channel;
	const int outputSize = output.cols * output.rows * channel;
	unsigned char *d_input, *d_output;

	// Allocate device memory
	cudaMalloc<unsigned char>(&d_input,inputSize);
	cudaMalloc<unsigned char>(&d_output,outputSize);

	// Copy data from OpenCV input image to device memory
	cudaMemcpy(d_input,input.ptr(),inputSize,cudaMemcpyHostToDevice);

	// Specify block size
	const dim3 block(BLOCK_SIZE,BLOCK_SIZE);

	// Calculate grid size to cover the whole image
	const dim3 grid((output.cols + block.x - 1)/block.x, (output.rows + block.y - 1)/block.y);

	// Start time
	cudaEventRecord(start);

	// Run Median Filter kernel on CUDA
	dilationFilter<<<grid,block>>>(d_input, d_output, output.cols, output.rows, channel);

	// Stop time
	cudaEventRecord(stop);

	//Copy data from device memory to output image
	cudaMemcpy(output.ptr(),d_output,outputSize,cudaMemcpyDeviceToHost);

	//Free the device memory
	cudaFree(d_input);
	cudaFree(d_output);

	cudaEventSynchronize(stop);
	float milliseconds = 0;

	// Calculate elapsed time in milisecond
	cudaEventElapsedTime(&milliseconds, start, stop);
	cout<< "\nProcessing time on GPU for dilation: " << milliseconds << "\n";
}
```

```
%%writefile programs/gaussianFilter.cu
```

```
#include "opencv2/imgproc/imgproc.hpp"
#include <opencv2/highgui.hpp>
#include <iostream>
#include <string>
#include <stdio.h>
#include <cuda.h>
#include "cuda_runtime.h"
#include <cmath>

#define BLOCK_SIZE      16
#define FILTER_WIDTH    8
#define FILTER_HEIGHT   8

using namespace std;

// Run Gaussian Filter on GPU
__global__ void gaussianFilter(unsigned char *srcImage, unsigned char *dstImage, unsigned int width, unsigned int height, int channel, float sigma) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    __shared__ float gaussianKernel[FILTER_WIDTH * FILTER_HEIGHT];

    if (threadIdx.x < FILTER_WIDTH && threadIdx.y < FILTER_HEIGHT) {
        int index = threadIdx.y * FILTER_WIDTH + threadIdx.x;
        gaussianKernel[index] = exp(-(threadIdx.x * threadIdx.x + threadIdx.y * threadIdx.y) / (2 * sigma * sigma)) / (2 * M_PI * sigma * sigma);
    }

    __syncthreads();

    if (x >= FILTER_WIDTH / 2 && x < width - FILTER_WIDTH / 2 && y >= FILTER_HEIGHT / 2 && y < height - FILTER_HEIGHT / 2) {
        for (int c = 0; c < channel; c++) {
            float sum = 0.0f;
            int index = 0;
            for (int ky = -FILTER_HEIGHT / 2; ky <= FILTER_HEIGHT / 2; ky++) {
                for (int kx = -FILTER_WIDTH / 2; kx <= FILTER_WIDTH / 2; kx++) {
                    int srcX = x + kx;
                    int srcY = y + ky;
                    sum += srcImage[(srcY * width + srcX) * channel + c] * gaussianKernel[index];
                    index++;
                }
            }
            dstImage[(y * width + x) * channel + c] = sum;
        }
    }
}

// The wrapper to call Gaussian filter
extern "C" void gaussianFilter_GPU_wrapper(const cv::Mat& input, cv::Mat& output, float sigma) {
    // Use cuda event to catch time
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    // Calculate number of image channels
    int channel = input.step / input.cols;

    // Calculate number of input & output bytes in each block
    const int inputSize = input.cols * input.rows * channel;
    const int outputSize = output.cols * output.rows * channel;
    unsigned char *d_input, *d_output;

    // Allocate device memory
    cudaMalloc<unsigned char>(&d_input, inputSize);
    cudaMalloc<unsigned char>(&d_output, outputSize);

    // Copy data from OpenCV input image to device memory
    cudaMemcpy(d_input, input.ptr(), inputSize, cudaMemcpyHostToDevice);

    // Specify block size
    const dim3 block(BLOCK_SIZE, BLOCK_SIZE);

    // Calculate grid size to cover the whole image
    const dim3 grid((output.cols + block.x - 1) / block.x, (output.rows + block.y - 1) / block.y);

    // Start time
cudaEventRecord(start);

// Run Gaussian Filter kernel on CUDA
gaussianFilter<<<grid, block>>>(d_input, d_output, output.cols, output.rows, channel, sigma);

// Stop time
cudaEventRecord(stop);

// Copy data from device memory to output image
cudaMemcpy(output.ptr(), d_output, outputSize, cudaMemcpyDeviceToHost);

// Free the device memory
cudaFree(d_input);
cudaFree(d_output);

// Synchronize the stop event
cudaEventSynchronize(stop);
float milliseconds = 0;

// Calculate elapsed time in milliseconds
cudaEventElapsedTime(&milliseconds, start, stop);
cout << "\nProcessing time on GPU for Gaussian filtering: " << milliseconds << "\n";
}
```

```
%%writefile programs/main.cpp
#include "opencv2/imgproc/imgproc.hpp"
#include <opencv2/highgui.hpp>
#include <iostream>
#include <string>
#include <stdio.h>
using namespace std;
```

```
extern "C" bool meanFilter_GPU_wrapper(const cv::Mat& input, cv::Mat& output);
extern "C" bool dilation_GPU_wrapper(const cv::Mat& input, cv::Mat& output);
extern "C" bool gaussianFilter_GPU_wrapper(const cv::Mat& input, cv::Mat& output);

// Program main
int main( int argc, char** argv ) {

  // name of image
  string image_name = "sample";
  //string image_name = "walk";

  // input & output file names
  string input_file = image_name+".jpeg";
  string output_directory = "generatedImages/";
  string output_file_meanFilter = output_directory + image_name+"_meanFilter.jpeg";
  string output_file_dilation = output_directory + image_name+"_dilation.jpeg";
  string output_file_gaussianFilter = output_directory + image_name+"_gaussianFilter.jpeg";

  // Read input image
  cv::Mat srcImage = cv::imread(input_file , cv::IMREAD_UNCHANGED);
  if(srcImage.empty())
  {
    std::cout<<"Image Not Found: "<< input_file << std::endl;
    return -1;
  }
  cout <<"\ninput image size: "<<srcImage.cols<<" "<<srcImage.rows<<" "<<srcImage.channels()<<"\n";

  // Declare the output image
  cv::Mat dstImage (srcImage.size(), srcImage.type());

  // run mean filter on GPU
  meanFilter_GPU_wrapper(srcImage, dstImage);

  // Output image after mean filter
  imwrite(output_file_meanFilter, dstImage);

  //run dilation on GPU
  dilation_GPU_wrapper(srcImage, dstImage);

  // Output image after dilation
  imwrite(output_file_dilation, dstImage);

  //run gaussianFilter on GPU
  gaussianFilter_GPU_wrapper(srcImage, dstImage);

  // Output image after gaussianFilter
  imwrite(output_file_gaussianFilter, dstImage);

  return 0;
}
```